

深入浅出 WebAssembly

于航 / 著



電子工業出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

WebAssembly 是一种新的二进制格式，它可以方便地将 C/C++ 等静态语言的代码快速地“运行”在浏览器中，这一特性为前端密集计算场景提供了无限可能。不仅如此，通过 WebAssembly 技术，我们还可以将基于 Unity 等游戏引擎开发的大型游戏快速地移植到 Web 端。WebAssembly 技术现在已经被计划设计成 W3C 的标准，众多浏览器厂商已经提供了对其 MVP 版本标准的支持。在 Google I/O 2017 大会上，Google 首次针对 WebAssembly 技术进行了公开演讲和推广，其 Post-MVP 版本标准更是对诸如 DOM 操作、多线程和 GC 等特性提供了支持。WebAssembly 所带来的 Web 技术变革势不可挡。

本书力求从一些简单的实践入手，深入理论，到复杂的具有实际业务价值的综合实践，深入浅出地介绍 Wasm 技术发展至今，其背后所涉及的各种底层设计原理与实现、相关工具链以及未来发展方向等多方面内容。本书内容包括：WebAssembly 技术的发展历程，从 PNaCl 到 ASM.js 再到 WebAssembly，以及这些技术的基本应用方法与性能对比；WebAssembly 的标准上层 API、底层堆栈机的设计原理，以及对 MVP 标准理论的深入解读；与 WebAssembly 标准相关的进阶内容，如单指令多数据流（SIMD）、动态链接（DL）等；LLVM 工具链与 WAT 可读文本格式的相关内容；基于 Emscripten 工具链开发 WebAssembly 应用的基本流程，以及工具链的一些基本常用功能和特性；基于 Emscripten 工具链实现 C/C++ 语言动态关系绑定技术；Emscripten 工具链所提供的一些如 WebGL 支持、虚拟文件系统、应用优化以及 HTML 5 事件系统等高级应用特性；构建一个具有实际业务价值的 WebAssembly 应用，现阶段 Wasm 生态的发展情况，以及在 Post-MVP 标准中制订的一些 WebAssembly 未来发展规划。

本书的目标读者为 Web 前端开发人员、C/C++ 开发人员和 WebAssembly 技术感兴趣的人员。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

深入浅出 WebAssembly / 于航著. —北京：电子工业出版社，2018.12
ISBN 978-7-121-35217-1

I. ①深… II. ①于… III. ①编译软件 IV. ①TP314

中国版本图书馆 CIP 数据核字（2018）第 238873 号

策划编辑：张春雨

责任编辑：葛 娜

印 刷：

装 订：

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编：100036

开 本：787×980 1/16 印张：34.25 字数：741 千字

版 次：2018 年 12 月第 1 版

印 次：2018 年 12 月第 1 次印刷

定 价：128.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zltts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819，faq@phei.com.cn。

序言（一）

I'm very excited to see this book, which covers in great detail a wide range of topics regarding WebAssembly. At this point in time WebAssembly is around one year old - if we count from when it shipped in all major browsers - so it's still fairly young, and the industry is just starting to figure out how revolutionary it is going to be. The potential is there for huge impact, and good documentation is necessary for that.

Why is WebAssembly's potential impact so large? For several reasons:

- **WebAssembly helps make the Web fast:** WebAssembly is designed for small download size, fast startup, and predictably fast execution. The improvement compared to JavaScript can be very significant, over 2x in many cases, and especially in startup, where the speedup can be 10x.
- **WebAssembly makes the Web competitive with native:** WebAssembly is designed as a compiler target for multiple languages. That includes C and C++, and in many areas of software the best implementations are in those languages, for example, game engines like Unity and Unreal, design software like AutoCAD, etc. It would take many years to write comparable products in JavaScript; instead, by compiling them to WebAssembly, the Web can be on par with native platforms today.
- **WebAssembly also fills an industry need outside the Web:** WebAssembly is fast, portable, sandboxed, has multiple excellent open source implementations, and just like the Web itself it is an industry standard expected to be supported for the long term. As a result, it's not surprising that WebAssembly is starting to be used outside of browsers, for example in the blockchain and content delivery network (CDN) spaces.

Looking back, it's remarkable that our industry has gotten to this point. Just a few years ago, there was no cross-browser collaboration on getting native code to run on the Web. Instead, there were multiple options, including Native Client, Adobe Alchemy, and ASM.js, each with its own advantages and disadvantages. I believe it was the momentum of ASM.js that got the industry to focus on fixing

things: ASM.js started out in Firefox, and by virtue of being a subset of JavaScript it immediately ran in all browsers - just not as efficiently. That led top companies in the video game industry and elsewhere to adopt ASM.js, together with Emscripten, the open source compiler to JavaScript that I started in 2010, and which could emit ASM.js. That adoption led to ASM.js support in Edge and later Chrome, at which point there was consensus that the industry should produce a proper standard in this space, which turned into WebAssembly. As the spec was designed and implementations started to appear, we added WebAssembly support to Emscripten, which allowed people to compile to both ASM.js and WebAssembly by just flipping a switch, making it easy for people to use the new technology. Finally, as of May 2018 Emscripten emits WebAssembly by default, and today WebAssembly has robust and stable support both in all major browsers and in the toolchain projects that emit it.

It's been a complicated path to get here, but the future looks bright. It is especially worth noting that WebAssembly is expected to add features like multithreading, SIMD, GC, and others, which will open up even more interesting opportunities.

Alon Zakai

Alon is a researcher at Mozilla, where he works on compile-to-Web technologies. Alon co-created WebAssembly and ASM.js, and created the Emscripten and Binaryen open source projects which are part of the primary WebAssembly compiler toolchain.

译文:

我很高兴能够看到这本书的出版,作者在书中详细地介绍了有关 WebAssembly 的各种主题。在这本书即将出版之际,WebAssembly 差不多一岁了一——如果从所有主流的 Web 浏览器开始支持 WebAssembly 算起,那么这项技术仍然相当年轻,业界也才刚刚开始意识到它将多么具有革命性。WebAssembly 所拥有的潜力将会在未来对 IT 行业产生巨大的影响,但在此之前,我们需要有优秀的文档。

为什么 WebAssembly 的潜在影响力会如此之大?有以下几个原因。

- WebAssembly 让 Web 应用运行更快。WebAssembly 是一种新的格式,文件体积更小,启动速度更快,运行速度也更快。与使用 JavaScript 构建的 Web 应用相比,性能提升非常明显。在大部分情况下,运行速度提升两倍以上,特别是在启动速度方面,速度提升

可以达到 10 倍。

- WebAssembly 让 Web 应用能够与原生应用展开竞争。WebAssembly 是多种编程语言的编译器目标，包括 C 和 C++。基于这些编程语言实现的优秀软件，如游戏引擎 Unity、Unreal，设计软件 AutoCAD 等，如果使用 JavaScript 开发在功能上与这些软件旗鼓相当的产品可能需要很多年时间。但如果将它们编译成 WebAssembly，这些原生应用就可以直接运行在 Web 平台上。因此，Web 能够与原生平台相提并论。
- WebAssembly 还在 Web 领域之外为行业带来了其他可能性。WebAssembly 运行速度快、可移植，提供了沙箱机制，并拥有众多优秀的开源实现，就像 Web 本身一样，它将会是一个被长期支持的行业标准。因此，WebAssembly 开始被应用在 Web 浏览器之外的领域也就不足为奇了，例如区块链和内容分发网络（CDN）。

回首过去，我们的行业能够取得如此的成就已经很了不起了。几年前，还没有人去进行这种跨浏览器协作，以便让原生代码运行在 Web 平台上。不过有很多不同的项目，如 Native Client、Adobe Alchemy 和 ASM.js，它们都在尝试做同样的事情，只是每个项目都有各自的优缺点。而我认为，是 ASM.js 的出现让业界开始专注于解决这个问题——ASM.js 最初出现在 Firefox 中，由于它是 JavaScript 的一个子集，因此可以无缝地运行在所有浏览器中，但运行效率不高。视频游戏等行业的一些顶级的公司开始尝试使用 ASM.js 和 Emscripten（我在 2010 年开源的编译器工具链，可以将代码编译成 ASM.js）。由于在这些领域的广泛应用，Edge 以及后来的 Chrome 均开始支持 ASM.js。此时，人们一致认为这个领域需要一个行业标准，于是 WebAssembly 出现了。

随着规范设计和实现的不断演进，我们在 Emscripten 中加入了 WebAssembly 支持——只需要在编译命令中加入一个“开关”，便可选择性地将编译目标设置为 ASM.js 或 WebAssembly，从而可以更轻松地使用这项新技术。截至 2018 年 5 月，Emscripten 已经将默认的编译目标类型改为 WebAssembly。今天，WebAssembly 已经在所有主流浏览器和工具链项目中得到了强大而稳定的支持。

一路走来历经坎坷，但未来是光明的。特别值得注意的是，WebAssembly 将会在未来添加多线程、SIMD、GC 等功能，而这些新特性将会为我们带来更多有趣的可能性。

Alon Zakai

Alon 是 Mozilla 的研究员，从事与“编译到 Web 平台”相关的研究工作。Alon 参与制定了 WebAssembly 和 ASM.js 标准，并创建了 Emscripten 和 Binaryen 等开源项目，这些项目都是 WebAssembly 编译器工具链的重要组成部分。

序言（二）

前端的可玩性变得越来越高，也越来越开放了。现如今，我们不仅仅能够使用 HTML、CSS 及 Javascript 来编写各种跨端的应用程序，WebAssembly 的出现还让我们能够以极小的成本来复用其他领域已存在的成果，以此来弥补 JavaScript 在其性能与功能上的不足。

我第一次了解到 WebAssembly 是在 2017 年年初，当时沉迷于想自己制作一个基于 Node.js 环境和树莓派的语音助手。可惜对于语音处理这个领域来说，JavaScript 还是一个“新人”，大量成熟的实现成果主要集中在 C/C++ 领域。因此，对于当时对 Node.js 扩展及 C/C++ 了解甚少的我来说，这是难度颇大的一个门槛。后来通过 Twitter 我了解到 WebAssembly 的前身是 ASM.js，于是我立即尝试使用 Emscripten 将 Google Assistant 的 Linux SDK 编译为 ASM.js，并顺利地在 Node.js 环境中进行了调用，那份喜悦我记忆犹新，同时这也极大地提升了我对这项技术的信心和好感。之后在全民直播的技术提升项目中，我与另一位研发人员有幸一起对最核心的播放器组件编解码和弹幕协议加密部分进行 WebAssembly 化，并成功上线且获得了极大的性能提升。在此之后，我坚信 WebAssembly 在未来一定会大有可为。

由于 WebAssembly 是一项极新的技术，因此在最初学习 WebAssembly 的过程中常常觉得知识零碎且不成体系，经常会出现浮沙驻高塔的情形，感觉入门十分困难。好在本书的出版，让这种情形不再复现。这本书的好处就是它系统详细地讲述了 WebAssembly 的方方面面，由浅入深地构建了整个 WebAssembly 的知识体系。不管你是刚接触 WebAssembly 的新人，还是已经在工作场景中使用 WebAssembly 的“老鸟”，通过阅读这本书都能够得到极大的提升。总之，如果你想了解 WebAssembly，或者想补足相关的知识体系，它都是一本不可多得的案头好书。风雨欲来，如果现在还不进行 WebAssembly 的技术储备，更待何时？

最后，我要感谢于航让我第一时间读到如此精彩的作品，同时也感谢他对 WebAssembly 在国内的布道普及所做的工作，我相信 WebAssembly 的未来一定会更加美好，Web 的未来也会更加开放和美好。

赵洋

赵洋是“全民直播”的前端研发经理，曾经主导全民直播播放器编解码核心模块及弹幕协议加密过程 WebAssembly 化。

前言

为什么要写这本书

自从 JavaScript（后面简称 JS）脚本语言于 1995 年诞生以来，人们便一直在使用该语言以及 HTML 与 CSS 来编写和开发以浏览器为主的 Web 应用。近年来，随着 JS 的不断流行，以及 Node.js 的出现，JS 也开始逐渐向除 Web 前端之外的其他领域发力，比如开发后端应用、机器学习应用乃至硬件编程等领域。但就 JS 本身而言，所不能无视的是它是一种弱类型语言，因此，相比于 C/C++ 等强类型语言，尽管 Chrome V8、SpiderMonkey 等 JS 引擎已经通过诸如 JIT 等多种技术手段来优化 JS 脚本代码的整体执行效率，但引擎每一次版本优化的迭代速度（所花费的时间）却远远跟不上当今各类 Web 应用的复杂程度变化。因此，发明一种能够从根本上解决该问题的技术便显得迫在眉睫。

曾昙花一现的 ASM.js、NaCl 与 PNaCl 等技术都尝试以其各自的方式来优化 Web 应用的执行效率，但由于其所存在的诸如“浏览器兼容性不佳”以及“性能优化不彻底”等问题，导致它们最终并没有被广泛推广。而在 2015 年出现的 WebAssembly（简称 Wasm）技术，便是在吸取了前者经验教训的基础上而被设计和发明出来的。现在，我们可以看到该项技术所具有的潜力——W3C 成立了专门的 WWG 工作组来负责 Wasm 技术的标准迭代与实现，四大主流浏览器（Google Chrome、FireFox、Edge 和 Safari）已经全部实现 WebAssembly 技术在其 MVP 标准中制定的所有特性，C/C++、Go 和 Rust 等高级语言已经逐渐开始支持编译到 Wasm 格式。这一系列的发展和变化都说明了人们对该项技术所寄予的厚望。

如今世间百态，万物的发展速度越来越快，而前端技术领域也同样如此，正在转向技术融合的道路——从 2000 年专门指代 PC 网页技术的 Web 前端，到 2010 年左右包含有 H5 技术的前端，再到融合了移动端甚至部分后端技术的“大前端”。“前端”一词所指代的技术实体正变得越来越模糊，已经不单单是指我们所熟知的 JS、HTML 与 CSS 了，正如大学里生物专业与化学专业两者融合后所形成的“生物化学专业”一样。技术本身并无好坏之分，只有能否适用于某些业务场景。而技术的融合则正好能够发挥各项技术本身所具备的优势，达到“1+1>2”的效果。WebAssembly 技术便正是如此。

在写作本书的过程中，笔者曾与 WWG 的核心成员 Alon、Ben 和 JF 等专家进行了多次交

流，以力求保证书中各个技术细节的正确性。但 Wasm 技术发展非常之快，比如 Emscripten 工具链每天都有众多的“commit”被提交到主分支中，新版本的发布也是以“周”甚至“天”为单位进行的。因此，书中所述内容并不保证会在今后的半年甚至一年时间里都具有时效性。而对于相关内容的时效性变化，笔者也会在对应于本书的 Github 仓库（详见“勘误和支持”部分）中及时进行标注。作为国内第一本介绍 WebAssembly 的技术书籍，希望本书的内容能够为国内互联网基础技术的发展做出微小的贡献。虽然现阶段我们还无法完全地自主创新，或者参与到各项国际技术标准的制定过程中，但唯有紧跟其脚步，才能够伺机超越。

本书特色

作为市面上第一本深入介绍 WebAssembly 技术的相关书籍，笔者尝试由浅入深地来介绍与 Wasm 技术相关的各种底层理论知识，以及相关编译器工具链的内部实现结构与使用方法。WebAssembly 技术从其第一版 MVP 标准诞生至今，时间过去并不久，但抽象的英文官方文档并不适合各类 Web 前端开发工程师直接进行阅读。从另一个方面来看，虽然我们可以在国内如“百度”等中文搜索引擎上找到部分与 Wasm 实践相关的介绍文章，但它们大都不会深入该技术标准的背后，探寻该技术的底层设计本质。因此，本书力求从一些简单的实践入手，深入理论，再到复杂的具有实际业务价值的综合实践，深入浅出地介绍 Wasm 技术发展至今，其背后所涉及的各种底层设计原理与实现、相关工具链以及未来发展方向等多方面内容。

由于 WebAssembly 并不是一种单方面的前端或后端技术，因此在本书中，我们将会随着内容的深入逐渐接触到除 Web 前端技术之外的诸如编译原理、V8 引擎、LLVM 以及 Linux 动态链接等多方面内容。笔者将会用最简单和直观的方式，由浅入深地介绍这些平日里可能很少接触到的技术与特性。

另外，作为市面上首本与 WebAssembly 相关的纯技术类书籍，笔者只能从自己所接触到的 Wasm 相关技术中，按照各个知识点的相关性与重要性来编排内容。相信读者在读完整本书后，一定会对 Wasm 技术背后的实现原理以及相关技术有进一步的理解。

读者对象

- Web 前端开发人员。
- C/C++开发人员。
- 对 WebAssembly 技术感兴趣的人员。

本书内容

本书分为 8 章。

第 1 章：本书的开篇，主要介绍 WebAssembly 技术的发展历程，从 PNaCl 到 ASM.js 再到 WebAssembly，以及这些技术的基本应用方法与性能对比。

第 2 章：介绍 WebAssembly 的标准上层 API、底层堆栈机的设计原理，以及对 MVP 标准理论的深入解读。

第 3 章：介绍与 WebAssembly 标准相关的进阶内容，如单指令多数据流（SIMD）、动态链接技术等。

第 4 章：由浅入深地介绍 LLVM 工具链与 WAT 可读文本格式的相关内容。

第 5 章：从理论走向实践，从本章开始介绍基于 Emscripten 工具链开发 WebAssembly 应用的基本流程，以及工具链的一些基本常用功能和特性。

第 6 章：介绍基于 Emscripten 工具链实现的 C/C++ 语言动态关系绑定技术。

第 7 章：从基础走向深入，继续介绍 Emscripten 工具链所提供的一些如 WebGL 支持、虚拟文件系统、应用优化以及事件系统等高级应用特性。

第 8 章：构建一个具有实际业务价值的 WebAssembly 应用，并介绍现阶段 Wasm 生态的发展情况，以及在 Post-MVP 标准中制订的一些 WebAssembly 未来发展规划。

勘误和支持

作为市面上首本介绍 WebAssembly 相关技术的书籍，本书在内容组织与编排上全部由笔者一人完成。由于笔者的知识水平有限，以及编写时间仓促，书中难免会出现一些错误或不准确的地方，恳请读者批评指正。如果你有更多的宝贵意见，欢迎在笔者的 Github 仓库“Book-DISO-WebAssembly”下创建“issue”，留下你的问题、意见或建议，笔者会在第一时间给予答复。

致谢

首先要感谢的是 Emscripten 工具链的作者 Alon，在编写 Emscripten 实践部分时，通过邮件

与他沟通了很多技术细节上的内容。Alon 是 Mozilla 的技术研究员，他主要负责维护和开发 Emscripten 与 Binaryen 两个重要的 WebAssembly 工具链。Alon 平日工作繁忙，非常感谢他能够抽出时间为我指导技术，以及本书内容组织方面的事情，并为本书写了序言。

其次要感谢的是我的女朋友，工作本已十分繁忙，而写书路漫漫，这期间曾遇到过无数次难以下笔、不知所措的时刻，而她却总是能在关键时刻鼓励我陪我共渡难关。在写书的半年多时间里，基本上所有的双休日都是在家中或咖啡馆度过的，没有时间陪伴，希望能够和你一同见证本书出版的时刻。

还要感谢的是我的好友以及家人。早早地立下写书的誓言，正是因为有了你们每天的监督，我才能督促自己完成每天规定的任务，最终完成本书所有章节的编写。

最后要感谢的是本书的策划编辑张春雨，2017 年上海 QCon 全球软件开发大会之后，他通过微信找到了作为讲师的我，并给予我这次写书的机会。但十分抱歉的是，由于我的日程和时间安排不当，导致本书的出版比预定时间晚了近三个月。但张老师并没有放弃我，这里真的要再说一声：“感谢您对我的信任”。

关于源码

关于本书第 3 章、第 4 章和第 5 章中所涉及的相关源代码示例，读者可以在笔者 Github 账号 (<https://github.com/Becavalier>) 下的“Cinderella”和“Book-DISO-WebAssembly”仓库中找到完整的代码文件。其他章节中所涉及的部分源代码文件，也会被同时整理并放置到上面的“Book-DISO-WebAssembly”仓库中。

轻松注册成为博文视点社区用户 (www.broadview.com.cn)，扫码直达本书页面。

- **提交勘误：**您对书中内容的修改意见可在 [提交勘误](#) 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动：**在页面下方 [读者评论](#) 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/35217>



目 录

第 1 章 漫谈 WebAssembly 发展史	1
1.1 JavaScript 的发展和弊端	1
1.1.1 快速发展与基准测试	1
1.1.2 Web 新时代与不断挑战	8
1.1.3 无法跨越的“阻碍”	11
1.1.4 Chrome V8 引擎链路	17
1.2 曾经尝试——ASM.js 与 PNaCl	28
1.2.1 失落的 ASM.js	28
1.2.2 古老的 NaCl 与 PNaCl	42
1.3 新的可能——WebAssembly	57
1.3.1 改变与颠覆	57
1.3.2 一路向前，WCG 与 WWG	85
第 2 章 WebAssembly 核心原理（基于 MVP 标准）	90
2.1 应用与标准 Web 接口	90
2.1.1 编译与初始化	90
2.1.2 验证模块	106
2.1.3 遇到错误	106
2.1.4 内存分配	108
2.1.5 表	112
2.2 深入设计模型——堆栈机	118
2.2.1 堆栈式虚拟机	119
2.2.2 逆波兰表达式	125
2.2.3 Shunting-yard 算法	126
2.2.4 标签与跳转	130
2.2.5 条件语句	135
2.2.6 子程序调用	137
2.2.7 变量	138

2.2.8	栈帧	139
2.2.9	堆	140
2.3	类型检查	141
2.3.1	数据指令类型	142
2.3.2	基本流程控制	144
2.3.3	基于表达式的控制流	149
2.3.4	类型堆栈的一致性	151
2.3.5	不可达代码	155
2.4	二进制编码	156
2.4.1	字节序——大端模式与小端模式	157
2.4.2	基于 LEB-128 的整数编码	161
2.4.3	基于 IEEE-754—2008 的浮点数编码	163
2.4.4	基于 UTF-8 的字符串编码	167
2.4.5	模块数据类型	168
2.4.6	虚拟指令与编码	169
2.4.7	类型构造符	174
2.5	模块	175
2.5.1	段	175
2.5.2	索引空间	185
2.5.3	二进制原型结构	186
2.6	内存结构	196
2.6.1	操作运算符	197
2.6.2	寻址	197
2.6.3	对齐	198
2.6.4	溢出与调整	203
第 3 章	动态链接与 SIMD（基于 MVP 标准）	204
3.1	动态链接（Dynamic Linking）	204
3.1.1	ELF	206
3.1.2	符号重定向（Symbol Relocation）	212
3.1.3	GOT（Global Offset Table，全局偏移表）	225
3.1.4	PLT（Procedure Lookup Table，过程查询表）	229
3.1.5	基于表的 Wasm 模块动态链接	233

3.2 单指令多数数据流 (SIMD)	237
3.2.1 SIMD 应用	239
3.2.2 并行与并发	243
3.2.3 费林分类法	244
3.2.4 SIMD.js & TC39	246
3.2.5 WebAssembly 上的 SIMD 扩展	248
第 4 章 深入 LLVM 与 WAT	250
4.1 LLVM——底层虚拟机	250
4.1.1 传统的编译器架构	251
4.1.2 LLVM 中间表示层	252
4.1.3 基于 LLVM 的编译器架构	254
4.1.4 LLVM 优化策略	256
4.1.5 LLVM 命令行工具	261
4.1.6 WebAssembly 与 LLVM	268
4.2 基于 LLVM 定义新的编程语言	272
4.2.1 图灵完备与 DSL	276
4.2.2 简易词法分析器	280
4.2.3 RDP 与 OPP 算法	287
4.2.4 AST (抽象语法树)	295
4.2.5 简易语法分析器	296
4.2.6 生成 LLVM-IR 代码	303
4.2.7 链接优化器	307
4.2.8 编译到目标代码	308
4.2.9 整合 I/O 交互层	312
4.3 WAT	315
4.3.1 S-表达式	316
4.3.2 WAT/Wasm 与 Binary-AST	318
4.3.3 其他与设计原则	320
第 5 章 Emscripten 基础应用	321
5.1 利器——Emscripten 工具链	321
5.1.1 Emscripten 发展历史	321
5.1.2 Emscripten 组成结构	323

5.1.3	Emscripten 下载、安装与配置	325
5.1.4	运行测试套件	329
5.1.5	编译到 ASM.js	330
5.2	连接 C/C++ 与 WebAssembly	332
5.2.1	构建类型	333
5.2.2	Emscripten 运行时环境	341
5.2.3	在 JavaScript 代码中调用 C/C++ 函数	350
5.2.4	在 C/C++ 代码中调用 JavaScript 函数	362
第 6 章	基于 Emscripten 的语言关系绑定	381
6.1	基于 Embind 实现关系绑定	383
6.1.1	简单类	388
6.1.2	数组与对象类型	390
6.1.3	高级类元素	392
6.1.4	重载函数	406
6.1.5	枚举类型	407
6.1.6	基本类型	408
6.1.7	容器类型	410
6.1.8	转译 JavaScript 代码	412
6.1.9	内存视图	415
6.2	基于 WebIDL 实现关系绑定	416
6.2.1	指针、引用和值类型	419
6.2.2	类成员变量	421
6.2.3	常量 “const” 关键字	422
6.2.4	命名空间	423
6.2.5	运算符重载	424
6.2.6	枚举类型	425
6.2.7	接口类	428
6.2.8	原始指针、空指针与 void 指针	430
6.2.9	默认类型转换	433
第 7 章	探索 Emscripten 高级特性	436
7.1	加入优化流程	436
7.1.1	使用编译器代码优化策略	441

7.1.2	使用 GCC 压缩代码	443
7.1.3	使用 IndexedDB 缓存模块对象	445
7.1.4	其他优化参数	452
7.2	使用标准库与文件系统	453
7.2.1	使用基于 musl 和 libc++ 的标准库	454
7.2.2	虚拟文件系统结构	457
7.2.3	打包初始化文件	459
7.2.4	基本文件系统操作	460
7.2.5	懒加载	469
7.2.6	Fetch API	473
7.3	处理浏览器事件	478
7.3.1	事件注册函数	479
7.3.2	事件回调函数	480
7.3.3	通用类型与返回值类型	481
7.3.4	常用事件	483
7.4	基于 EGL、OpenGL、SDL 和 OpenAL 的多媒体处理	486
7.4.1	使用 EGL 与 OpenGL 处理图形	487
7.4.2	使用 SDL 处理图形	493
7.4.3	使用 OpenAL 处理音频	496
7.5	调试 WebAssembly 应用	499
7.5.1	编译器的调试信息	499
7.5.2	使用调试模式	501
7.5.3	手动跟踪	502
7.5.4	其他常用编译器调试选项	504
第 8 章	WebAssembly 综合实践、发展与未来	505
8.1	DIP 综合实践应用	505
8.1.1	应用描述	505
8.1.2	滤镜与卷积	506
8.1.3	基本组件类型与架构	510
8.1.4	编写基本页面骨架 (HTML 与 CSS)	511
8.1.5	编写核心卷积函数 (C++)	512
8.1.6	编写主渲染循环与“胶水”代码 (JavaScript)	514

8.1.7	使用 Emscripten 编译并运行应用	519
8.1.8	性能对比	520
8.2	WebAssembly 常用工具集	521
8.2.1	Cheerp	521
8.2.2	Webpack 4	523
8.2.3	Go 和 Rust 的 WebAssembly 实践	525
8.2.4	Binaryen	528
8.2.5	WasmFiddle	529
8.2.6	Wabt	530
8.2.7	AssemblyScript	530
8.3	WebAssembly 未来草案	530
8.3.1	GC（垃圾回收）	531
8.3.2	Multi-Thread（多线程）与原子操作	531
8.3.3	异常处理	531
8.3.4	多返回值扩展	531
8.3.5	ES 模块	531
8.3.6	尾递归	532
8.3.7	BigInts 的双向支持	532
8.3.8	自定义注释语法	532

第 1 章

漫谈 WebAssembly 发展史

佛曰：“世间一切事物，皆有缘而来”。如果将这句话翻译成更加直白且通俗易懂的语句，那就是“世间遇到的一切事物，都是有其各自出现的理由的”。而本书接下来将要介绍的“WebAssembly”也并不例外。自 1969 年全球互联网（Internet）的诞生，到 1991 年世界首款 Web 浏览器的出现，再到 2018 年全球网民数量突破 40 亿人，我们看到的是 IT 技术的飞速发展，以及信息时代的一次又一次迭代变更。而在接下来的几年里，随着互联网技术的又一次不断革新，名为“WebAssembly”的技术将会作为众多新技术中的新星，引领着 Web 应用走向下一个发展的新时代。

1.1 JavaScript 的发展和弊端

一直以来，在 Web 前端领域我们都主要使用 JavaScript 语言来编写运行在浏览器上的 Web 应用。不仅如此，随着 React Native、Electron 及 Vue.js 等用于各种目的的开发框架的出现，JavaScript 语言正变得越来越流行，直至一跃成为 Github 语言排行榜的年度冠军。但反观如今的各类 Web 应用，其功能逐渐复杂化，对性能的要求逐渐提高。而浏览器作为运行平台，虽然其内部的 JavaScript 引擎也在不断被优化，但限于 JavaScript 语言本身的一些特性，根本无法满足日益增长的应用性能需求。

1.1.1 快速发展与基准测试

自 1997 年 ECMAScript 1.1 版本标准作为一个正式草案被提交给欧洲计算机制造商协会（ECMA），使得 ECMAScript 这种脚本语言规范开始逐渐走向标准化，一直到 2017 年 6 月，ECMAScript 2017（ES 8）标准的正式发布，ECMAScript 标准的不断发展带动着 JavaScript 这门以其作为标准实现的脚本语言不知不觉地走过了 20 个年头。

在这 20 个年头里，不只是 ECMAScript 标准本身在语法和特性上有了翻天覆地的变化，包括用来解析和执行 JavaScript 脚本语言的 Web 浏览器、用来快速进行前端构建的各种前端 JavaScript 框架、基于 Chrome V8 用来进行服务端应用开发的 Node.js 运行时引擎，甚至是那些用于辅助前端应用开发的包括各种构建和自动化工具在内的与 JavaScript 相关的开源软件，都有了十足的发展。

首先值得一提的是 JavaScript 这门脚本语言在编程语法和功能特性上的发展。自 1997 年 ECMAScript 标准诞生一直到 2015 年，在这十几年时间里，ECMAScript 标准本身并没有做过太多的改动，只是在不断调整语言稳定性的过程中，使语义更加严格，同时还增加了少许的新特性。1999 年发布的 ECMAScript 3 版本增加了对“正则表达式”的支持，而在这其后的整整十年内，ECMAScript 标准一直处于一个相对的平稳期，其间没有做任何修订和改动。直到十年后的 2009 年，在 ECMAScript 5 标准中又增加了“严格模式”，以及对“JSON”这种轻型数据交换格式进行编/解码等操作的支持（除此之外，还有少许其他新特性）。在六年后的 2015 年，ECMAScript 标准迎来了史上最大的一次在语言特性上的改动，那就是 ECMAScript 6 标准的诞生。ECMAScript 6 标准提供了很多丰富的新特性和语法糖，包括从 Python 借鉴来的迭代器和生成器特性、集合类型、箭头函数、类型数组，以及可以用于面向 OOP 编程的“类”关键字和用于元编程的“代理”特性等。自 ECMAScript 6 标准发布开始，在接下来的两年内 ECMAScript 标准又相继发布了 ECMAScript 2017 和 ECMAScript 2018 版本，同时 ECMAScript 标准也开始以发布年份作为版本号重新命名，从此 ECMAScript 标准也进入了每年一次版本迭代的快速发展新时代。ECMAScript 标准的快速发展也同样促使前端开发领域跟着快速发展，一大批前端 JavaScript 框架和前端技术架构体系也由此诞生（这里 ECMAScript 6 对应 ECMAScript 2015）。

从十年前擅长直接操作 DOM 对象的 jQuery 到十年后以 MVVM 模式架构见长的 React 以及 Vue.js 等框架，前端框架的发展突飞猛进，不断地改变着人们在前端应用领域的日常开发模式。从最初碎片组件化的开发模式到现在已经逐渐成体系化、完整组件化和分层次的开发模式，前端开发的效率也在不断提升。当然，前端框架的日益增多也离不开 JavaScript 这门编程语言所应用的领域变得愈加广泛。现在 JavaScript 语言可以被应用在 Web 前端开发、移动端 APP 开发、服务端应用开发、桌面端应用开发、深度学习应用开发甚至是硬件开发等多个领域。不仅如此，自基于 V8 构建的 Node.js 引擎出现后，JavaScript 语言也正逐渐“力挽狂澜”，变得“无所不能”。

通过统计近十年来（从 2008 年 1 月 1 日到 2017 年 11 月 1 日）全球开发者每年在 Github 上所创建的 JavaScript 开源项目的数量（如图 1-1 所示），可以发现，每年所创建的 JavaScript 开源项目的数量与前一年相比几乎都呈指数型增长。毋庸置疑，2015 年是 JavaScript 最流行的

一年，从 2014 年到 2016 年的三年时间里，Angular.js、React 和 Vue.js 这三个 Web 前端开发框架“巨头”相继出现，在 Web 前端开发领域呈分庭抗礼之势。

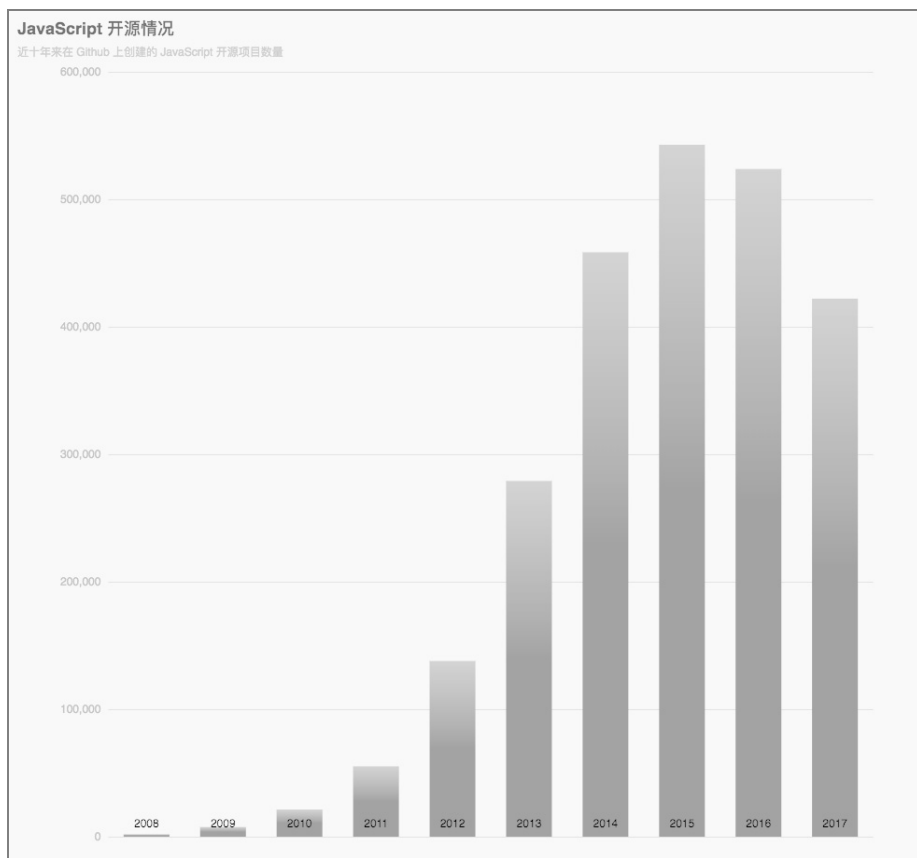


图1-1 近十年来Github上基于JavaScript语言构建的开源项目数量

当然，JavaScript 之所以能够深入到如此众多行业和领域的开发实践当中，也离不开 Web 浏览器在性能上的日益优化和提升。以 Chrome 浏览器的核心 JavaScript 引擎 V8 为例，如图 1-2 所示，从官方 Github 仓库中每日对其修改提交数量统计数据中可以看到，每天都有 30~50 个提交被合并到 V8 项目的主分支。V8 的完整版本号由四位数字组成，形式为“MAJOR.MINOR.BUILD.PATCH”。如某一个 V8 版本的版本号为“6.3.292.33”，其中第三位数字对应的“BUILD”字段在每次 V8 重新编译和发布后都会增长。而事实上，该字段对应数字在 V8 每天的小版本发布中都会有 10 次以上的增长，可见其版本发布之频繁，性能优化和特性迭代速度之快。

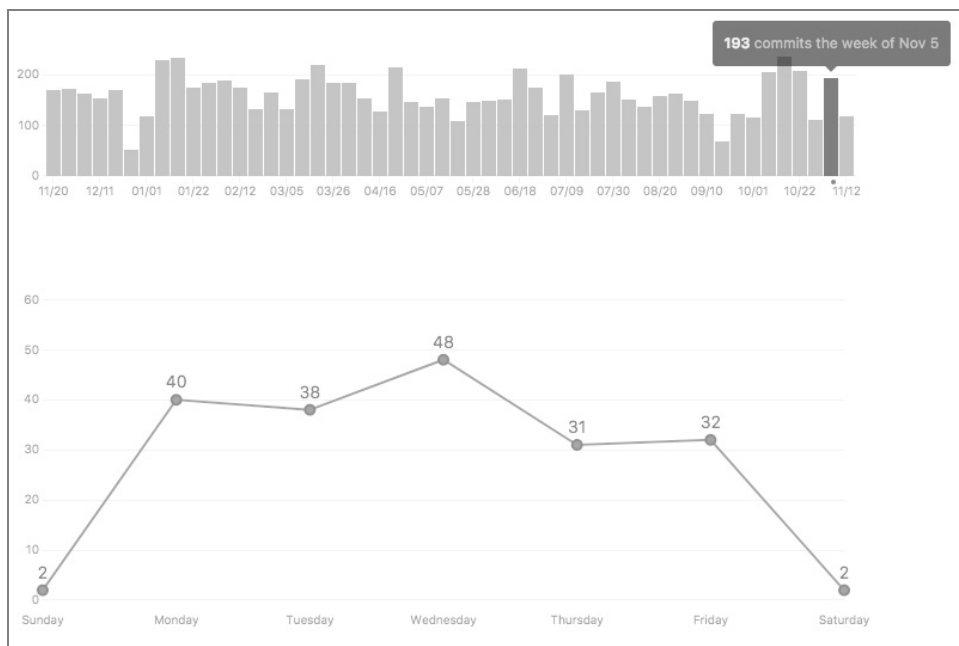


图1-2 Github上V8项目的每日修改提交数量统计图

那么现如今的 Web 浏览器对 JavaScript 代码的性能优化到底达到了怎样的程度呢？我们以下面所给出的代码为例，来看一下同样一段耗时的业务逻辑，在不同编程语言的对应实现下，其代码的运行效率和性能表现如何。这里我们还是以 Chrome 的最新版本桌面浏览器作为用来测试该 JavaScript 应用的容器，采用的 Chrome 版本是 62.0.3202.94 (Official Build) (64-bit)。用于测试的应用其业务逻辑是：初始化两个浮点数类型的变量，然后让其中一个变量的值等于该值与另一个变量的累加和，并重复该过程 1 亿次，程序会打印出这 1 亿次累加操作所花费的时间，并重复进行 10 次。最后再将这 10 次的 1 亿次累加和打印出来。

首先，我们基于原生 C++ 语言编写应用，该应用对应的具体源代码如下：

```
benchmark-cpp.cc
#include <stdio.h>
#include <ctime>

int main() {
    // 声明两个双精度浮点类型的变量
    double a = 6.245614, b = 2.718;
    int i, j;
    clock_t start, end;
```



```

for(j = 0; j < 10; j++) {
    // 记录累加开始前的时钟周期
    start = clock();
    for(i = 0; i < 1000000000; i++) {
        a = a + b;
    }
    // 记录累加结束后的时钟周期
    end = clock();

    // clock() 函数会返回程序运行至此时 CPU 已经走过的时钟周期数
    // CLOCKS_PER_SEC 表示每秒钟 CPU 走过的时钟周期数
    printf("Time Cost: %lums\n", (end - start) * 1000 / CLOCKS_PER_SEC);
}
// 打印结果
printf("a = %lf\n", a);
return 0;
}

```

在 C++ 源代码编写完毕之后，需要通过编译器来编译这段源代码，将其转换成一个二进制的可执行文件。这里将编译的程序分为两个版本，第一个为不经过任何编译器优化处理的基本版本；第二个为经过编译器对 C++ 源代码进行浮点数优化和代码优化后生成的版本。

1. 未经过编译器优化的版本，编译、链接与运行

```
g++ benchmark-cpp.cc -o benchmark-cpp
```

运行所生成的程序

```
./benchmark-cpp
```

2. 编译器对代码和浮点数操作优化后的版本

```
g++ -O3 -ffast-math benchmark-cpp.cc -o benchmark-cpp
```

运行所生成的程序

```
./benchmark-cpp
```

可以看到，这是一个用来计算浮点数累加值的程序。该程序一共循环 10 次，每一次都会打印出浮点数累加 1 亿次后所花费的时间。程序在最后会打印出循环 10 次后，即累加 10 亿次后得到的浮点数变量值。这里在进行 C++ 代码编译时分别编译出了两个不同版本的程序，第一个是未经过任何编译器优化直接生成的版本；而在第二个版本中，我们加入了 GCC 编译器支持的，可以对浮点数运算进行优化的参数“-fast-math”来优化代码中的浮点数运算。同时还指定了编译器需要对 C++ 代码进行优化的等级参数“-O3”。

接下来，我们继续编写与该段程序业务逻辑相同的其他语言版本的程序代码。首先给出

JavaScript 版本代码，这段使用 JavaScript 语言编写的代码可以直接在 Chrome 开发者模式下的 Console（控制台）中运行。

```
(function() {
  var a = 6.245614, b = 2.718;
  var i, j;
  for(j = 0; j < 10; j++) {
    // 记录当前的时间（从时间原点算起）
    var start = performance.now();
    for(i = 0; i < 100000000; i++) {
      a = a + b;
    }
    d2 = new Date();
    // 打印经过的时间
    console.log(`(performance.now() - start) ms`)
  }
  console.log("a = " + a);
})();
```

然后继续编写 Java 版本的代码，这段代码仍然基于同样的业务逻辑。

```
benchmark-java.java
```

```
import java.util.Date;

class Benchmark {
  public static void main(String args[]){
    double a = 6.245614;
    double b = 2.718;
    long start, end;
    for (int j = 0; j < 10; j++) {
      start = new Date().getTime();
      for (int i = 0; i < 100000000; i++) {
        a = a + b;
      }
      end = new Date().getTime();
      System.out.printf("Time Cost: %d ms\n", (end - start));
    }
  }
}
```

在编译上述代码时请确保本地环境已经安装了 Java 开发工具包（JDK）。可以在命令行下

运行如下命令来编译该 Java 程序。

```
# 编译代码
javac benchmark-java.java
# 运行代码
java benchmark
```

最后给出的是 Python 语言对应的代码。这里采用的 Python 解释器版本是 2.7.13，将下面的代码直接存储在一个以“.py”为后缀的文件中，然后直接使用“python”命令在命令行下执行该文件即可。

```
benchmark-python.py
import time

def main():
    a = 6.245614
    b = 2.718
    for i in xrange(10):
        start = time.time()
        for j in xrange(100000000):
            a = a + b
        end = time.time()
        print("%f ms" % ((end - start) * 1000))

    print("a = %f\n" % a)

if __name__ == "__main__":
    main()
```

通过 Python 解释器在命令行下运行 Python 文件中的代码。

```
# 解释执行
python benchmark-python.py
```

我们对上述 4 种不同编程语言分别对应的 5 段程序的运行结果进行了统计，并计算出了每段程序在 10 次“大循环”中消耗的平均时间值。最后我们将统计结果绘制成如图 1-3 所示的柱状图（前 10 列为 5 段程序对应的 10 次“大循环”过程的单次统计结果，最后一列为 10 次“大循环”的平均统计结果）。

从图 1-3 所示的基准测试平均结果中可以看到，运行效率最高的是经过编译器对代码进行优化和浮点数操作优化后的 C++ 程序，单次循环的平均耗时只有 3ms。其次便是基于 Java 和 JavaScript 编写的程序，两种程序在处理单次“大循环”时的耗时基本相同，平均时间均为 100ms

左右，不相上下。而在相同的程序业务逻辑下，未经过任何编译器优化处理的 C++程序的运行效率并不是很高，基准测试结果显示其运行效率低于在同样业务逻辑下使用 JavaScript 和 Java 语言编写的程序，平均耗时为 300ms 左右。排名最后的是使用 Python 语言编写的程序，同样的单次循环平均耗时在 6000ms 以上（测试数据仅供参考）。

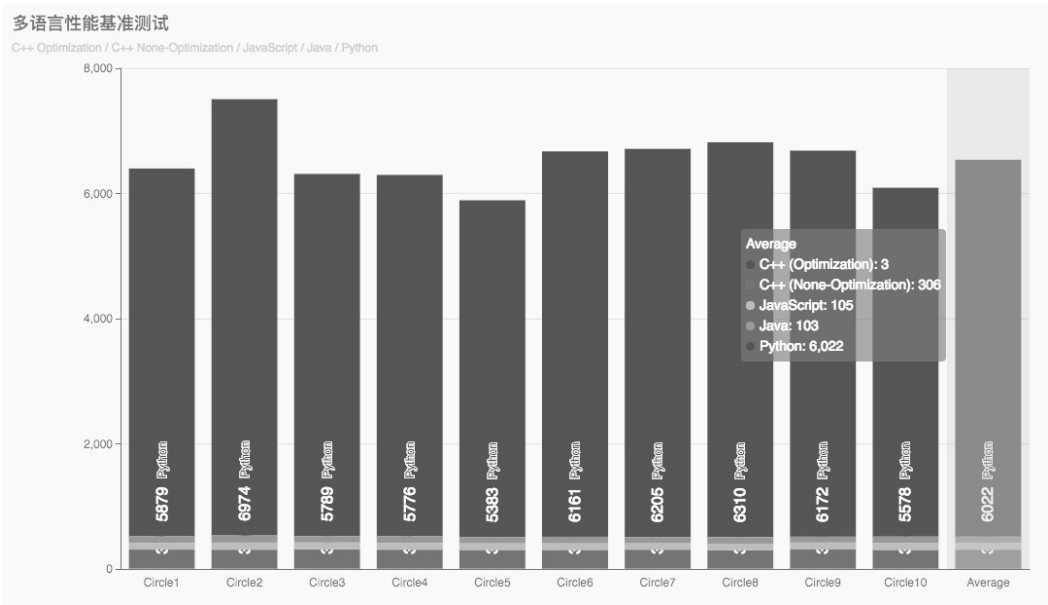


图1-3 多语言性能基准测试的统计结果

综合来看，JavaScript 代码在现代 Web 浏览器中的解析和运行效率其实并不低，虽然相比经过编译器优化的 C/C++代码来说还有一些差距，但在日常开发工作中经常接触到的那些编程语言里，JavaScript 代码的解析和运行效率已经处于比较高的水平。现实总是残酷的，虽然 JavaScript 代码的解析和运行效率正在随着 JavaScript 引擎的不断优化改进而不断提升，但在日常工作中我们所遇到的前端应用也正变得越来越复杂和多样化。

1.1.2 Web 新时代与不断挑战

一般来说，一项新技术是否会随着时代的推进而被快速迭代和发展，要看这项技术所应用的实际业务场景中是否有相应的技术需求，毕竟没有任何技术会被凭空创造出来，那么技术的迭代和发展速度也就取决于这些业务需求的变化速度。技术决定了业务需求的多样性，而业务需求的多样性又反过来推动技术本身不断向前发展，两者相辅相成，最终才能推动行业整体的发展和进步。

自 1991 年 HTTP 协议和 HTML（超文本标记语言）这两种核心的 Web 技术诞生以来，Web 技术领域便开始不断发生翻天覆地的变化。如图 1-4 所示为 1991—2002 年 Web 技术的总体发展情况，在这十二年里，Web 技术的总体发展过程还是比较缓和和稳定的。首先是 NetScape、Opera 和 Internet Explorer（IE）三大浏览器开始逐渐走入人们的视野。一些用于构建更丰富 Web 应用的基础性技术开始逐渐涌现，比如 Flash 技术从 1996 年开始可以被应用在浏览器端，这使得我们可以在传统的 Web 应用中嵌入包含丰富多媒体信息的 Flash 应用，这一发展也使得 Web 应用的交互性和动态性大大增强。Flash 技术的出现催生了一批以提供视频播放、视频发布和视频分享服务为主的视频服务平台，同时也推动了基于 Flash 的 Web 页游行业的发展。

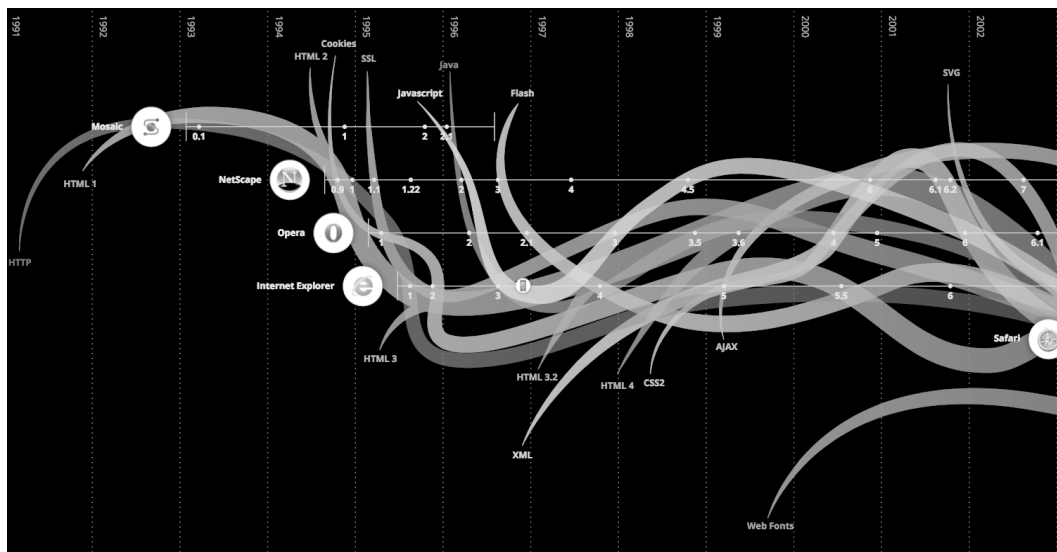


图1-4 1991—2002年Web技术的总体发展情况

从 2002 年开始，Web 技术的发展便到了其整个发展历程的“下半场”。如图 1-5 所示为 2003—2012 年这十年间 Web 技术的总体发展情况，在这十年的时间里，新型 Web 技术的出现逐渐呈现出爆炸式的增长。首先是 Chrome、Firefox 和 Safari 这三种为推动 Web 技术的爆炸式发展做出了巨大贡献的浏览器开始出现，各大浏览器厂商对其产品的版本更新迭代速度也开始加快。Web 技术从 2008 年开始便进入了一个爆炸式的快速发展阶段，各种各样的新型 Web 浏览器特性，以及新的 Web 标准和 ECMAScript 语言标准都如雨后春笋般开始涌现出来。XMLHttpRequest2 技术为 Web 应用的数据传输提供了更加方便和高效的方式；WebRTC 技术为 Web 应用的实时在线视频 / 语音直播提供了底层的基础技术解决方案；WebGL 技术为 Web 应用提供了一种可以通过 JavaScript 来使用 Web 浏览器版 OpenGL 的特性，基于 WebGL 暴露

出的 JavaScript 接口, 我们可以在 Web 网页上高效地绘制 3D 动画和模型, 而这为基于 Web 网页运行大型 3D 网络游戏提供了可能; IndexedDB 技术为前端应用的结构化数据存储和高性能检索提供了支持。除此之外, 还有很多的 Web 相关技术正在或已经被实现和标准化, 这些技术无疑都大大地拓宽了 Web 应用所能够覆盖到的应用领域和场景。也正是自 2008 年 HTML 5 标准和 2009 年 CSS 3 标准出现后, Flash 多媒体应用技术在 Web 开发领域逐渐走下坡路, 直至最后被其他技术取代。由此可见 Web 领域的技术迭代与更替速度之快。

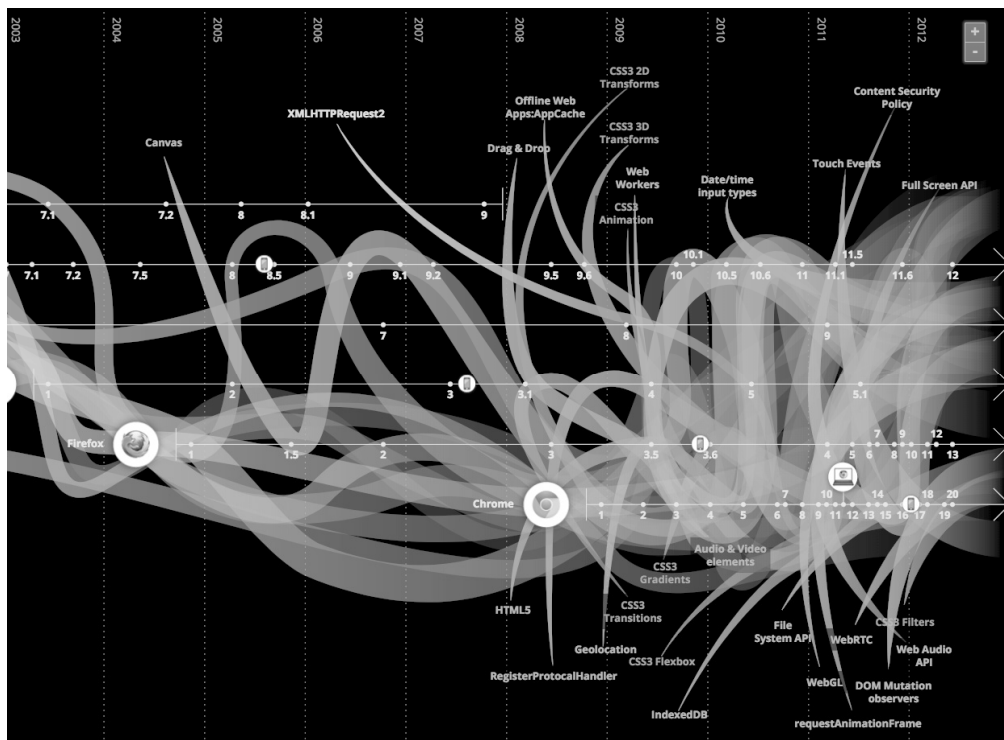


图1-5 2003—2012年Web技术的总体发展情况

JavaScript 作为一门可用于开发 Web 前端应用的编程语言, 从 1995 年发展至今, 其所能够应用的领域已经不再局限于最原始的、基于浏览器的 Web 应用开发。包括 Node.js 在内的一系列新出现的 JavaScript 运行时环境已经把 JavaScript 语言的应用场景从前端应用开发带到了服务器端的应用开发。

基于 Chrome V8 引擎构建的 Node.js 和 Fib.js 等 JavaScript 运行时环境, 为后端服务器应用的开发提供了非阻塞的异步 IO 和基于事件模型等新特性, 这些新特性可以让我们以开发传统前端 Web 应用的思路来开发服务器端应用。不仅如此, 基于 Node.js 开发出来的各种服务器端应

用框架更是极大地提高了我们开发后端应用的效率。这些框架在一些必要的业务流程上已经为我们做了足够多的封装和优化，这使得我们可以更多地去关注业务逻辑代码的实现，而不是一些底层的架构细节。但事情并没有这么完美，以 Node.js 为例，由于其本身是基于 V8 实现的，而 V8 最重要的一个功能就是对 JavaScript 代码进行解析和优化，然后将优化好的中间代码编译成机器码或其他格式后再进行处理。因此，无论 Node.js 对 V8 上层的 JavaScript 代码进行了何种底层系统调用流程上的优化，如果最后 V8 在解析和优化 JavaScript 代码的过程中消耗了大量时间，那么整个应用的运行效率必然会大打折扣。总的来说，Chrome V8、JavaScriptCore 和 SpiderMonkey 等 JS 引擎对 JavaScript 代码的解析和优化效率，直接决定了基于 JavaScript 开发的前端和服务器端应用的运行效率，进而也影响了产品的用户体验。

除此之外，变得日益复杂和庞大的 Web 前端应用也带来了更多对 JavaScript 语言性能上的挑战。比如基于 Web 浏览器的视频处理应用、大型 3D 游戏以及在线的机器学习（深度学习）实时训练平台等，无一例外都需要消耗大量的浏览器计算资源，因此 JS 引擎对 JavaScript 代码的解析执行效率高低也直接决定了这些应用能否被流畅地运行。不仅如此，我们都知道通过 JavaScript 来移动或修改网页上的 DOM 节点所付出的成本是巨大的，随着传统 Web 页面的交互设计变得越来越复杂，这种成本损耗所带来的性能问题可能会被逐渐放大，这也是我们在未来将要面对的问题。

1.1.3 无法跨越的“阻碍”

前面我们提到，Chrome V8 和 JavaScriptCore 等 JS 引擎对 JavaScript 代码的解析和执行效率高低，会直接影响到那些基于 JavaScript 开发的前后端应用的运行效率，那么 JS 引擎在解析和执行 JavaScript 代码时究竟会在哪些地方消耗较多的时间和性能呢？下面让我们走进这些 JS 引擎，来看一下它们内部的“世界”。

```
function add (a, b) {  
    return a + b;  
}
```

上面给出了一段简单的 JavaScript 代码，在这段代码中我们定义了一个非常简单的 JavaScript 函数。调用该函数时一共需要传入两个参数，函数会直接返回这两个参数经过“+”运算符运算后的结果。那么当我们在代码中调用该函数时，JS 引擎会对这个函数的调用过程进行怎样的处理呢？如图 1-6 所示，这是在 ECMA-262 最新的 8.0 版本标准中规定的当 JavaScript 引擎遇到“+”运算符时需要进行的语法分析规则。

12.8.3 The Addition Operator (+)

NOTE The addition operator either performs string concatenation or numeric addition.

12.8.3.1 Runtime Semantics: Evaluation

AdditiveExpression : *AdditiveExpression* + *MultiplicativeExpression*

1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be ? GetValue(*lref*).
3. Let *rref* be the result of evaluating *MultiplicativeExpression*.
4. Let *rval* be ? GetValue(*rref*).
5. Let *lprim* be ? ToPrimitive(*lval*).
6. Let *rprim* be ? ToPrimitive(*rval*).
7. If Type(*lprim*) is String or Type(*rprim*) is String, then
 - a. Let *lstr* be ? ToString(*lprim*).
 - b. Let *rstr* be ? ToString(*rprim*).
 - c. Return the String that is the result of concatenating *lstr* and *rstr*.
8. Let *lnum* be ? ToNumber(*lprim*).
9. Let *rnum* be ? ToNumber(*rprim*).
10. Return the result of applying the addition operation to *lnum* and *rnum*. See the Note below 12.8.5.

图1-6 在ECMA-262标准中规定的对“+”运算符的语法分析规则

在这段标准中说明了 JavaScript 引擎需要对“+”运算符两边的操作数进行怎样的处理和转换。我们将这段比较抽象的标准“翻译”成较为容易理解的流程描述，说明如下。

首先，这段标准说明了“+”运算符两边可以使用的操作数类型。这个操作数可以来源于 *AdditiveExpression* 或 *MultiplicativeExpression* 表达式所对应的值。粗粗一看就可以发现，*MultiplicativeExpression* 表达式是指由“*（乘法）”、“/（除法）”、“%（取余）”及“**（求幂）”等运算符组成的一系列表达式；而 *AdditiveExpression* 表达式则是指由“+（加法）”和“-（减法）”运算符组成的表达式。但实际上，每一种类型的表达式都是从下到上由一连串的表达式“继承”链组成的。比如对于一个 *AdditiveExpression* 表达式，我们自上而下来推导它的继承链结构，可以发现一个独立的 *MultiplicativeExpression* 表达式本身也是一个 *AdditiveExpression* 表达式，而一个独立的 *ExponentiationExpression* 表达式同时也是一个 *MultiplicativeExpression* 表达式，依此类推，直到整个继承链的最底层。链路底层直接对应的是数字的实体类型，这些类型又会被分为 *NonZeroDigit* 及 *DecimalDigit* 等各种类型。

整个继承链路如图 1-7 所示。这些出现在 ECMAScript 标准中的各类型表达式之间复杂的继承关系，也决定了 JavaScript 引擎在解析 JavaScript 代码时应该如何确定各个运算符之间的运算优先级关系。

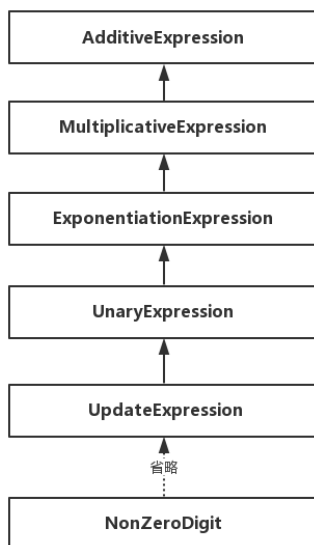


图1-7 在ECMA-262标准中部分表达式的继承链关系

通过上述说明我们可以了解到，“+”运算符与其两边的子表达式共同组成了一个新的 `AdditiveExpression` 表达式。接下来，我们可以按照标准给出的详细流程来进一步了解当 JS 引擎在解析 JavaScript 代码时，如果遇到“+”运算符应该以怎样的规则来解析这个新生成的 `AdditiveExpression` 表达式，并最终求得这个表达式的值。

JavaScript 引擎在进行语法 / 语义分析时，首先需要判断该“+”运算符两边子表达式的值，这个步骤对应于上述 ECMAScript 8.0 标准中的第 1 步和第 3 步，这里在解析 `AdditiveExpression` 子表达式值时进行的是一个递归的过程。然后通过标准给出的抽象方法 `GetValue` 来对“+”运算符两边已经解析好的操作数进行处理，这里的 `lval` 和 `rval` 分别代表两个处理好的结果值。接下来，继续通过抽象方法 `ToPrimitive` 对上一步中得到的 `lval` 和 `rval` 两个值进行处理，返回的 `lprim` 和 `rprim` 分别代表经过这一步处理后得到的两个中间结果值。在第 7 步中，我们需要判断上一步的两个结果值 `lprim` 和 `rprim` 中是否至少有一个值的类型为 `String`（字符串）。如果是，则分别对 `lprim` 和 `rprim` 两个值调用 `ToString` 抽象方法进行处理，然后将这两个处理后的结果值（分别对应 `lstr` 和 `rstr` 代表的字符串）拼接成一个完整的字符串，并将该字符串作为最终结果返回。如果不是，则会对 `lprim` 和 `rprim` 代表的中间结果值调用 `ToNumber` 抽象方法进行处理。这个抽象方法会按照相应的规则将两个中间结果值分别转换成对应的数字值，最后再将这两个数字值通过数学加法进行计算，并将计算结果返回。

这里需要注意的是，为了能够更加严谨地将“ECMA—262”标准直观地表达出来，我们还

需要对上述分析过程中的几个地方有更深刻的认识。

标准中的“变量”名

需要注意的是，我们在上文中提到的“lval”和“rval”等标记名称，实际上并不代表任何 JavaScript 引擎在其源码中实际使用到的变量名、寄存器名或组件名等，只是标准文档为了方便在描述解析流程时将各个阶段的“中间结果”表示出来而取的标记名称。而且这些标记名称也十分有规律，比如 lval 可以理解为 Left Value，即运算符左操作数所代表的值；lstr 代表 Left String，即左操作数对应的字符串值，其他可以依此类推。用这些标记名称来表示标准在各个处理阶段所生成的值类型显得十分贴切和形象。

抽象方法

我们在上文中提到的抽象方法，其实在 ECMAScript 8.0 标准中对应的名词是“Abstract Operations”。Abstract Operations 本身并不是 ECMAScript 语言的一部分，只是用来帮助标准本身来更加清晰地描述语法和语义。

这里称之为“抽象方法”，是由于 Abstract Operations 在标准文档中的书面引用形式与 JavaScript 语言中的函数调用过程十分类似。所谓的 Abstract Operations 其实本身也是一系列对给定值的特定处理流程。如图 1-8 所示为抽象方法 ToNumber 的处理流程规范。当 ToNumber 抽象方法遇到不同类型的操作数时会根据标准分别进行不同的处理，这里以比较复杂的 Object 类型操作数为例。对于一个 Object 类型的操作数，抽象方法 ToNumber 首先需要对该操作数调用抽象方法 ToPrimitive 进行处理，然后将其返回的结果再通过调用抽象方法 ToNumber 来进行处理并得到最终的处理结果。同样的，抽象方法 ToPrimitive 所对应的 Abstract Operation，也有其自己的一套规范化、标准化的处理流程。

Argument Type	Result
Undefined	Return NaN.
Null	Return +0.
Boolean	If <i>argument</i> is true , return 1. If <i>argument</i> is false , return +0.
Number	Return <i>argument</i> (no conversion).
String	See grammar and conversion algorithm below.
Symbol	Throw a TypeError exception.
Object	Apply the following steps: 1. Let <i>primValue</i> be ? ToPrimitive(<i>argument</i>, hint Number). 2. Return ? ToNumber(<i>primValue</i>).

图1-8 在ECMA-262标准中对抽象方法ToNumber的处理规范定义

非终结符、终结符与产生式

为了能够更加深入地理解 ECMAScript 规范中表达式的作用，以及表达式与 JavaScript 引擎之间的关系，我们可以将大多数编程语言所对应编译器的语法分析过程总结成如下通用的简化版流程。这里假设使用某种编程语言编写了如下一行代码。

```
thisIsAVariable = 1 + 2
```

这行代码表示一个最简单的赋值语句，将等号“=”右边的表达式结果值赋值给了一个名为“thisIsAVariable”的变量。我们使用字母“S”来表示整个赋值语句表达式，那么应该怎样通过符号组合的形式来描述这条赋值语句呢？假设使用字母“v”来表示赋值语句最左边的变量，字母“e”表示“=”运算符，字母“p”表示“+”加号运算符，字母“d”表示一个整数。经过整理，我们得到了如下所示的字符表达式，最右侧的五个字母从左至右依次对应上述语句中出现的每一个有效的语法元素。

```
S -> vedpd
```

但实际上，由于“=”运算符右边可以放置任意类型的表达式，因此我们继续对上述字符表达式进行修改，用字母“E”来表示一个任意类型的表达式。经过整理后的字符表达式如下。

```
S -> veE
```

在这里，我们将上述字符表达式称为赋值语句“S”的产生式。其中字母“v”和“e”所对应的 Token 类型已经确定。Token 是词法分析器在进行词法分析时产生的最小的且具有明确语义的有效关键字。比如对于上述赋值语句，在词法分析阶段，词法分析器会将这段代码按照最基本的关键字进行分割，分割出的“thisIsAVariable”、“=”、“1”、“+”和“2”五个字符串片段中每一个都独立地称作一个 Token，并且其各自分别对应着一种具体的 Token 元素类型。而这些能够直接与某类型 Token 相对应的符号，我们称它们为“终结符”。相反的，符号“E”可以表示一个具有任意组成元素的表达式类型，而其本身并没有被明确地指定与哪些 Token 相对应，因此我们称它为“非终结符”。

为了能够清楚地描述符号“E”所对应表达式的具体结构，我们需要对“E”进行名为“非终结符展开”的操作。为了简化该流程，假定在该编程语言中只存在“加法”这一种数学运算，同时也只有“整数”这一种数据类型。那么符号“E”所代表的表达式便可能具有两种组成方式，其中一种是符号“E”可以仅由一个整数数字面量组成，该整数独立地作为表达式完成整个“S”表达式的计算过程；另一种是符号“E”可以表示任意经由“+”加法运算符组合而形成的子表达式的值。因此，我们可以进一步对赋值语句“S”的产生式进行如下整理。

```
E -> d|Epd
```

```
S -> ve(d|Epd)
```

可以看到，符号“E”的产生式以递归形式表示出来。这种递归形式可以使该表达式的展开式能够覆盖到具有任意长度子表达式的所有具体表达式上。比如对于如下这种包含有连加运算的数学表达式，借助上述表达式“S”的递归形式展开式，我们可以通过以下步骤对其进行展开。

```
thisIsAVariable = 1 + 2 + 3;
```

展开过程

1. S -> veE
2. S -> veEpd (E->Epd)
3. S -> veEpdpd (E->Epd)
4. E -> vedpdpd (E->d)

通常来说，在编程语言所对应的整个编译器链路中，词法分析器（Lexer）负责将源代码中的各类短语进行过滤并解析成具有特定语义的 Token 字符串，而这些字符串将会在接下来的语法分析阶段，被语法分析器（Parser）通过相应的算法进行“表达式非终结符”展开的处理。比如在 JavaScript 语言中，我们可以通过“+”运算符来定位一个 AdditiveExpression 类型的表达式。刚才我们也提到过，每一个表达式其实都是一种“非终结符”类型，这些非终结符都需要在被编译成机器码之前展开成特定的终结符形式。因此相对而言，如果语法分析器无法将一段代码内的某个表达式展开成标准中提到的任意一种终结符展开式形式，那么在该表达式中便一定存在语法格式错误。

同样的，如果代码中不存在语法错误，也就代表着该段代码中所有的非终结符结构（包括表达式、条件控制结构及函数定义在内的各类非终结符形式）都被成功地展开成了标准中某种特定的终结符形式。语法分析器在处理完代码后会向编译器链路的下一个阶段输出一种名为“抽象语法树（AST）”的数据结构，它以结构化的表示形式表达了整段代码的语法结构。至此，也表明语法分析器真正“理解”了源代码中各个代码段所表达的具体语义。

通过上面的分析，我们大致了解了 JavaScript 引擎在处理“+”运算符时所需要经过的一系列解析流程。实际上，JS 引擎在实际实现规范中所描述的各类流程时，需要处理的细节问题远比我们所描述的要复杂得多。在规范中出现的每一个流程内的每一个抽象方法都有着其各自不同的处理流程，而在这些流程内部同样又有多个更加底层的抽象方法。通过将这些抽象方法一步一步组合形成各种各样的上层流程，而流程与流程之间又相互调用形成网状结构，这些网状结构的调用流程最后便组成了整个 ECMAScript 语法和语义层的标准。

回过头来看，JavaScript 引擎之所以要在处理“+”运算符时经过多道“工序”，其主要原因是 JavaScript 本身是一种弱类型（Weak Typed）的编程语言。所谓弱类型，在语法形式上最直观的体现便是在使用该编程语言初始化变量时，无须显式地指出变量的具体类型，整个变量的

类型完全由代码解释器在代码的运行过程中进行推断。而相对于弱类型编程语言的则是强类型（Strongly Typed）编程语言。同样的，所谓强类型，最直观的体现便是在使用该编程语言声明变量时，必须要显式地指明变量需要存储的数据类型。这样做的好处是，我们无须花费额外的精力在代码运行时去推断变量的数据类型，而这从某种程度上便可以大大提高代码的运行效率。由于代码中的所有变量类型都不再需要通过运行时环境去推断，因此便可以提前将程序的源代码进行静态编译（AOT）和优化，最后直接生成相应的经过优化的二进制机器码供 CPU 执行。C 语言便是这样一种常用的强类型编程语言。强类型编程语言所共有的一个优势就是无须进行变量的运行时类型推断，进而使得代码的运行效率更高。

1.1.4 Chrome V8 引擎链路

从上文中我们已经了解到，由于 JavaScript 引擎无法在代码运行前便得知变量所存储的具体数据类型，因此对于很多运算符操作，JS 引擎需要在运行时环境下通过一系列的判断“决策”才能推断出变量所存储的具体数据类型。而实施这些“决策”的过程则需要消耗一定的系统资源。接下来让我们以老版本（Chrome 58 以下）的 Chrome V8 引擎为例，来进一步剖析 V8 引擎在处理 JavaScript 源代码时的整个流程以及所对应的编译器链路，如图 1-9 所示。

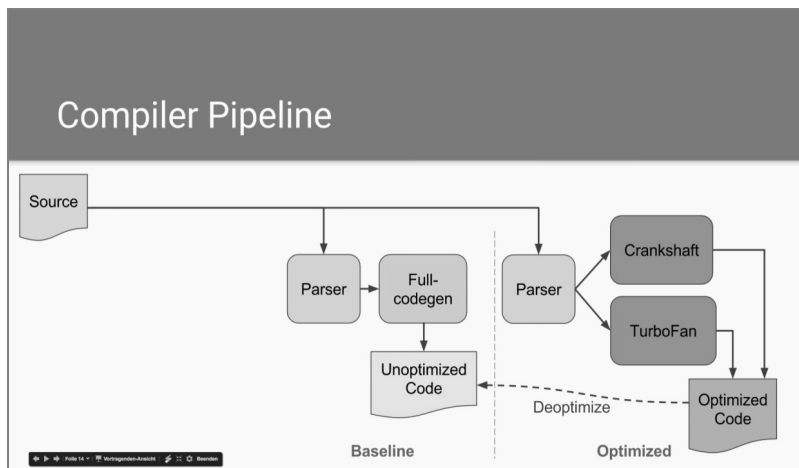


图1-9 老版本Chrome V8引擎的编译器链路

V8 引擎的飞跃式进化也是从这里开始的。整个代码的解析、编译和执行流程按照 JavaScript 代码所经过的不同编译器顺序及类型，可以被分为“Full-codegen”基线 JIT 编译器对应的 Baseline 编译阶段，以及“TurboFan”和“Crankshaft”两个优化 JIT 编译器所对应的 Optimized 编译阶段共两个部分。

首先，每一组编译器都会有一个前置的语法分析器，这个语法分析器会对 JavaScript 源代码进行词法和语法分析，然后生成对应的抽象语法树结构。一般来说，一个完整的编译器链路在处理源代码时通常会分为以下几个步骤。

词法分析

词法分析，顾名思义，该阶段主要是识别和提取源代码中的关键字，同时判断出每一种关键字的类型。这些关键字可能是一个用来声明变量的“let”，也可能只是一个简单的用于结束语句的分号“;”。所有这些组成源代码的关键字都会在这一阶段被识别和提取出来，最后结合这些关键字的具体类型和一些基本的语素信息，我们就得到了词法分析阶段的最小单位“Token”，而每一个 Token 就代表了一种不同的关键字类型。

语法分析

语法分析，该阶段会通过结合编程语言的具体语法规则和从词法分析阶段得到的 Token 信息，将 JavaScript 源代码转换成抽象语法树（AST）的形式。抽象语法树以树状的形式表示源代码的语法结构。

```
// 声明一个函数
function add(a, b) {
    return a + b;
}
// 声明一个变量并调用函数，最后将函数值赋值给该变量
let num = add(1, 2);
```

比如我们在上面的 JavaScript 源代码中声明了一个函数，这个函数接收两个参数，然后返回这两个参数经过“+”运算符运算后的结果。接下来调用该函数，并将数字 1 和 2 作为参数传入该函数，最后再将函数返回的结果赋值给一个新的变量“num”。我们将这段 JavaScript 源代码经过语法分析处理后生成的 AST 以 JavaScript 对象的形式表示出来，如下所示。这里我们采用了 Esprima 来分析 JavaScript 源代码并生成对应的 AST 结构。

```
{
  "type": "Program",
  "body": [
    {
      "type": "FunctionDeclaration",
      "id": {
        "type": "Identifier",
        "name": "add"
      },
    },
  ],
}
```

```
"params": [
  {
    "type": "Identifier",
    "name": "a"
  },
  {
    "type": "Identifier",
    "name": "b"
  }
],
"body": {
  "type": "BlockStatement",
  "body": [
    {
      "type": "ReturnStatement",
      "argument": {
        "type": "BinaryExpression",
        "operator": "+",
        "left": {
          "type": "Identifier",
          "name": "a"
        },
        "right": {
          "type": "Identifier",
          "name": "b"
        }
      }
    }
  ]
},
"generator": false,
"expression": false,
"async": false,
"leadingComments": [
  {
    "type": "Line",
    "value": "声明一个函数",
    "range": [
      0,
```

```
        10
      ]
    }
  ],
  "trailingComments": [
    {
      "type": "Line",
      "value": "声明一个变量并调用函数，最后将函数值赋值给该变量",
      "range": [
        52,
        80
      ]
    }
  ]
},
{
  "type": "VariableDeclaration",
  "declarations": [
    {
      "type": "VariableDeclarator",
      "id": {
        "type": "Identifier",
        "name": "num"
      },
      "init": {
        "type": "CallExpression",
        "callee": {
          "type": "Identifier",
          "name": "add"
        },
        "arguments": [
          {
            "type": "Literal",
            "value": 1,
            "raw": "1"
          },
          {
            "type": "Literal",
            "value": 2,
```



```

        "raw": "2"
      }
    ]
  }
},
"kind": "let",
"leadingComments": [
  {
    "type": "Line",
    "value": "声明一个变量并调用函数，最后将函数值赋值给该变量",
    "range": [
      52,
      80
    ]
  }
]
},
"sourceType": "script"
}

```

从上面的 AST 结构中可以看到，抽象语法树上的每一个节点都有其各自的类型，这些类型可能是 ECMAScript 标准中的一个终结符 Token，比如标识符（Identifier）类型；或者是一个非终结符，比如一个需要进一步展开的表达式类型。除此之外，还有函数定义、变量定义等各种类型的节点。在语法分析阶段，语法分析器会根据文法来分析那些在词法分析阶段产生的 Token，并且按照 ECMAScript 的语法规则将这些 Token 进行整理和组合，通过识别关键字 Token 来提取出语法中的函数定义、变量定义和表达式等语法结构。最后再将这些包含有各种元素的层级结构整理成一个树状的语法表示结构。AST 从最内层（树的子节点）开始统一由终结符类型的 Token 组成，逐层向外扩展。比如对于 CallExpression 表达式类型，其展开过程可以有多种形式，每种形式又由不同的 Token 组成，而具体的展开形式只有根据实际的源代码才能确定。同时 CallExpression 也可以被放到 VariableDeclaration 这种语法结构中。AST 便是这样逐层地将各种类型的语法元素按照标准中规定的语法结构表示出来的，树形结构的表示方式也正好对应了源代码在语法上的逻辑嵌套和组成关系。

语义分析

在语法分析阶段，编译器只是对源代码进行静态分析，以判断源代码在语法表达上是否存

在错误。在语义分析阶段，编译器会进一步分析在语法分析阶段产生的 AST 结构，进而判断源代码是否存在运行时错误。在这一阶段中，编译器会分析源代码中的函数调用过程，以及传入函数的参数个数是否正确，优化那些已经声明并被初始化，但却在程序中没有被明确调用到的变量等。对于强类型的编程语言，编译器还会检查变量类型的声明与使用是否保持一致。

生成目标代码

经过上面几个步骤之后，我们便可以将最终经过分析和优化后的代码直接“翻译”成对应的目标代码并在对应的目标平台上执行。比如 JavaScript 代码对应的目标执行平台就是各类浏览器。在这一阶段中，编译器会将从上一步语义分析阶段得到的中间代码直接编译成对应平台的机器码，并在浏览器中解析和执行。

我们再把目光移回到 V8 引擎上。为了提高对 JavaScript 源代码的解析和执行效率，V8 引擎会对当前即将执行的 JavaScript 代码段进行分析。如前面的图 1-9 所示，V8 引擎会首先将所有的 JavaScript 源代码通过一个前置的语法分析器（Parser）来进行词法和语法的分析，并同时生成对应的 AST 数据结构。在这一阶段中，Parser 会检查整段 JavaScript 代码并将它们分成如下两种不同的类型。

Top-Level 代码

这一类型的代码主要是指那些当 JavaScript 源代码初次加载时需要被首先运行到的“顶层”代码。这部分代码主要包括变量声明、函数定义以及函数调用等类型的代码。而那些用于定义函数的函数体内部的代码，并不属于 Top-Level 代码。

非 Top-Level 代码

这一类型的代码则与 Top-Level 代码正好相反，主要是指那些用于定义函数的函数体内部的 JavaScript 代码。

在如下的一段代码中，我们将其中的 Top-Level 代码和非 Top-Level 代码通过注释做出了区分。其中注释为 TL 的代码为 Top-Level 代码，而注释为 NTL 的代码为非 Top-Level 代码。

```
function add (a, b) {                                // TL
    return a + b;                                     // NTL
}

function abs (x) {                                    // TL
    return Math.abs(x);                               // NTL
}
```

```
function minusAbs (a, b) {           // TL
    let t = a - b;                   // NTL
    return abs(t);                   // NTL
}
let i = 10;                          // TL
console.log(minusAbs(i, 10));        // TL
```

在 V8 引擎中，位于各个编译器的前置 Parser 被分为 Pre-Parser 与 Full-Parser 两种类型。首先，Pre-Parser 主要负责对整个 JavaScript 源代码段进行必要的前期检查。Pre-Parser 并不区分具体的代码类型，即无论这些代码是否属于 Top-Level 类型，Pre-Parser 都会对它们进行检查。通过检查，V8 会判断 JavaScript 代码中是否存在语法错误，如果存在，则 V8 需要在对代码进行下一步处理前及时抛出语法错误信息（Early Syntax Error）并提示用户，同时中断代码的后续解析和执行。在 Pre-Parser 对代码进行分析和处理的阶段，Parser 并不会生成对应于源代码语法的 AST 结构，同时也不会生成变量可用的上下文作用域。

接下来，Full-Parser 会开始分析那些属于 Top-Level 类型的 JavaScript 源代码，并生成这部分 JavaScript 代码所对应的 AST 信息。同时在该阶段中，Full-Parser 还会对代码中的变量进行作用域分析，以便追踪那些具有特殊作用域的变量（例如闭包中的变量），并为它们在外层作用域分配相应的资源，同时生成该变量可用的上下文作用域。当 Full-Parser 将所有属于顶层 Top-Level 类型的 JavaScript 源代码都转换成对应的 AST 信息后，这些 AST 随后便会被送往 V8 引擎的第一个支持运行时编译（JIT）的编译器——“Full-codegen”基线编译器来进行处理。在这里，Full-codegen 会快速地根据输入的 AST 信息来编译并生成对应未经优化的机器码，这些机器码最后便可以被浏览器快速地解析和执行。

浏览器在解析和执行这些 Top-Level 代码的过程中，会遇到一些诸如函数调用的操作。由于在初次的 Full-Parsing 过程中，Parser 只对 Top-Level 代码进行了处理，而在这部分 Top-Level 代码中并不包含这些被调用函数的函数体定义。因此在这种情况下，V8 引擎会根据 Top-Level 代码在执行过程中遇到的函数调用，来对 JavaScript 源代码中对应函数的函数体再进行一次 Full-Parsing 的处理，并生成这个函数体所对应的 AST 信息。这些 AST 信息随后也同样会被 Full-codegen 处理并生成对应的机器码，最后再由浏览器解析和执行。V8 引擎的这种“不一次性完全生成和处理所有 JavaScript 源代码对应的 AST 信息，而只在用到时才进行 AST 生成和编译”的特性，我们称之为“Lazy Parsing”。总的来说，在 V8 引擎中，Pre-Parsing 阶段主要用来检查整个 JavaScript 源代码中是否含有需要提前抛出的语法错误，而在随后的 Full-Parsing 阶段才会真正生成 AST 信息并交由编译器来处理。

随着从 Full-codegen 基线编译器输出的未经优化的机器码被浏览器解析和执行，V8 引擎会发现在当前这些正在运行的代码逻辑中，有一些比较耗时的代码流程可以被进一步优化。比如在 JavaScript 代码中出现的“大次数循环代码块”或 ECMAScript 6 标准中的某些新特性。这时，V8 引擎会选择把这部分 JavaScript 代码直接转交给另外的优化编译器进行优化处理。V8 引擎有两个 JavaScript 优化编译器，分别为 Crankshaft 和 TurboFan。其中 Crankshaft 主要用于对 JavaScript 源代码进行一些比较基础的优化；而 TurboFan 主要用于对那些使用了 ECMAScript 6 及以上标准的新特性代码进行优化，同时它也负责对 ASM.js 代码进行处理。

首先，Full-codegen 基线编译器在根据 AST 来生成未经优化机器码的过程中，会假设某些情况是成立的。也只有当在确保这些假设是真实成立的情况下，优化编译器所进行的深度优化才会有实际的意义。比如在优化一段包含有大量循环逻辑的 JavaScript 代码时，如果 Full-codegen 基线编译器发现在循环的前几次中都执行了相同逻辑的代码（相同的作用域环境与变量结构），便会假设在后面的所有循环中，迭代的代码形式都是保持不变的，而对于这样的循环结构，基线编译器会把编译流程交给优化编译器来进行优化处理。但实际上，由于 JavaScript 语言本身所具有的高度动态性，所以并不能完全保证循环逻辑中每次迭代所执行的代码结构都一定是完全相同的。因此，这些经过优化编译器生成的机器码在被浏览器解析和执行前，V8 引擎会再次检验之前基线编译器所做出的假设是否成立。如果假设确实成立，浏览器便会直接解析和执行这些经过优化生成的机器码；如果假设不成立，优化编译器便会开始执行一个名为“去优化（Deoptimize）”的过程。

“去优化”过程是指 V8 引擎发现之前基线编译器所做出的假设并不成立，而此时则需要把代码的编译流程从优化编译器重新“交回”到基线编译器的手上。基线编译器会再次重新编译这些 JavaScript 代码，同时生成未优化的机器码，最后让浏览器来解析和执行。而之前优化编译器生成的那部分错误的优化机器码便会被直接丢弃。

以上便是老版本 Chrome V8 引擎在处理 JavaScript 代码时，从编译器角度来看所经过的一系列流程。随着 Web 应用的规模越来越大，V8 引擎现有的这种用于处理 JavaScript 代码的编译器架构模式所存在的问题便逐渐凸显出来。Full-codegen 基线编译器在处理 Top-Level 代码时所生成的机器码会大量占用 V8 的堆内存。不仅如此，V8 的编译器链路在解析和执行 JavaScript 代码的整个时间线（Startup Time）上，有近三分之一的时间被 Parsing 和 Compiling 占据。其中，对同一段代码的多次 Parsing 更是大大降低了 V8 引擎对 JavaScript 源代码的处理效率（Pre-Parsing、基线编译器的 Full-Parsing 和优化编译器的 Full-Parsing）。比如对于如下所示的这段 JavaScript 源代码，我们尝试通过 Node.js 来追踪 V8 引擎在处理该段代码时的 Pre-Parsing 和 Full-Parsing 过程。（注：Node.js 基于 V8 构建，因此也使用 V8 来解析 JavaScript 代码。）

```
app.js
function sayHi (name) {
  let message = "Hi " + name + "!";
  for (let i = 0; i < 100; i++) {
    console.log(message);
  }
}

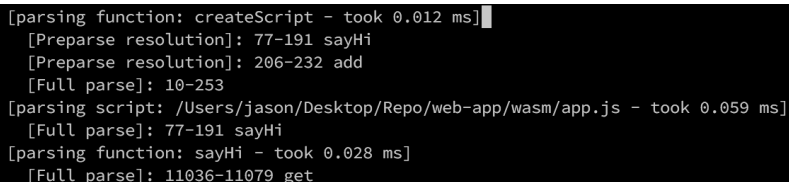
function add (a, b) {
  return a + b;
}

sayHi("Jason");
```

在终端命令行中执行如下 Node.js 命令。

```
node -trace_parse app.js
```

通过为 node 命令指定“-trace_parse”参数，我们可以让 Node.js 输出 V8 引擎在对 JavaScript 源代码进行 Pre-Parsing 和 Full-Parsing 两个过程时的相关信息，结果如图 1-10 所示。可以看到，在解析这段 JavaScript 源代码时，V8 引擎会首先对这段源代码中的所有函数定义及调用过程进行 Pre-Parsing 操作，在 Pre-Parsing 过程中 V8 引擎会检查源代码中是否存在需要提前抛出的语法错误信息。在对所有源代码完成 Pre-Parsing 阶段的处理后，V8 引擎开始处理在第一次运行时将会被调用到的函数代码，即属于 Top-Level 类型的代码。



```
[parsing function: createScript - took 0.012 ms]
[Preparse resolution]: 77-191 sayHi
[Preparse resolution]: 206-232 add
[Full parse]: 10-253
[parsing script: /Users/jason/Desktop/Repo/web-app/wasm/app.js - took 0.059 ms]
[Full parse]: 77-191 sayHi
[parsing function: sayHi - took 0.028 ms]
[Full parse]: 11036-11079 get
```

图1-10 通过Node.js来追踪上述JavaScript代码在V8下的Parsing过程

随着代码的运行，一些在 Top-Level 代码中被调用的函数其函数体会被 V8 引擎继续处理。这里由于我们在 Top-Level 代码中调用了 sayHi 函数，因此 V8 引擎会对 sayHi 函数的函数体再进行一次 Full-Parsing 处理，Full-Parsing 操作会分析函数体代码并生成所对应的 AST 数据结构，同时初始化函数内的变量和上下文环境。

但从整体上看，V8 引擎对 sayHi 函数的 Pre-Parsing 处理过程其实是完全没有必要的。由于所有位于 Top-Level 层级的函数调用都会在 JavaScript 被加载时就执行，对这些函数的函数体的

语法检查完全可以放在紧接着 Pre-Parsing 之后的 Full-Parsing 阶段来进行，如果可以将这些重复的操作完全省略掉，那么 V8 引擎在处理 JavaScript 代码时的效率又会得到进一步的提升。

为此，V8 引擎也提供了一些比较“Hack”的方式来避免多余的 Pre-Parsing 过程。我们可以通过一种常见的名为“IIFE（立即执行函数表达式）”形式的 JavaScript 代码，来强制让 V8 引擎省略掉对 IIFE 内部代码的 Pre-Parsing 过程。在这里，我们通过 IIFE 来优化之前的那段代码，优化后的 JavaScript 源代码如下。

```
app.js
var sayHi = (function (name) {
  let message = "Hi, " + name + "!";
  for (let i = 0; i < 100; i++) {
    console.log(message);
  }
})();

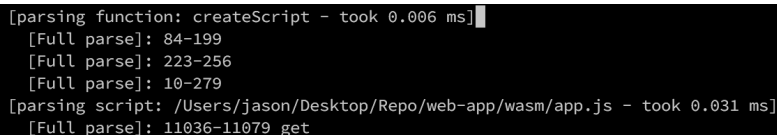
var add = (function (a, b) {
  return a + b;
})();

sayHi("Jason");
```

在终端命令行中运行如下 Node.js 命令。

```
node -trace_parse app.js
```

接下来，我们还是使用 Node.js 来追踪 V8 引擎对上述经过 IIFE 处理后的 JavaScript 代码进行 Pre-Parsing 和 Full-Parsing 的处理过程，命令执行结果如图 1-11 所示。可以看到，V8 引擎在处理这些包含在 IIFE 结构内的代码时，完全省略了对这部分代码的 Pre-Parsing 处理过程，这在某种程度上提升了 V8 引擎解析和执行 JavaScript 代码的效率。



```
[parsing function: createScript - took 0.006 ms]
[Full parse]: 84-199
[Full parse]: 223-256
[Full parse]: 10-279
[parsing script: /Users/jason/Desktop/Repo/web-app/wasm/app.js - took 0.031 ms]
[Full parse]: 11036-11079 get
```

图1-11 经过IIFE修改后的JavaScript代码执行结果

鉴于在老版本 V8 引擎中存在各种问题，Google 自 Chrome 58 版本开始，开始对 V8 引擎的编译器链路进行了改进与优化。V8 团队希望能够通过如图 1-12 所示的全新编译器链路架构模式来优化现阶段的 V8 引擎。

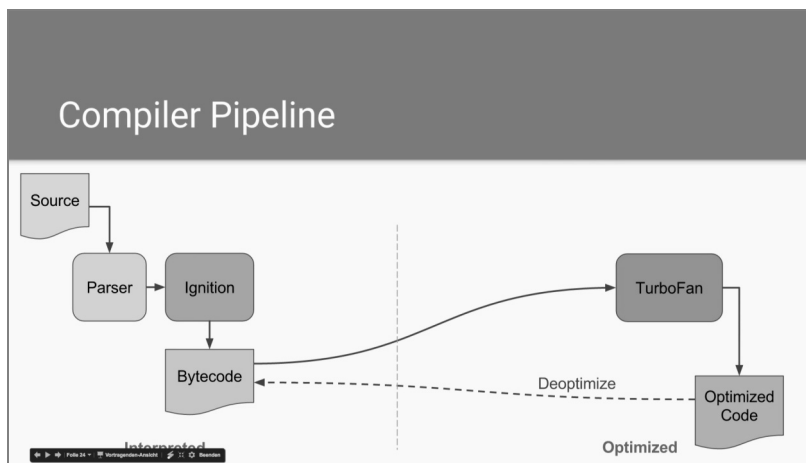


图1-12 全新的Chrome V8编译器链路

在这个全新的 V8 引擎编译器链路中，新加入了一个名为“**Ignition**”的解释器，同时去掉了 Full-codegen 基线编译器和 Crankshaft 优化编译器。**Ignition** 解释器会根据从 Parser 传递过来的基于 JavaScript 源代码生成的 AST 信息直接生成对应的“比特码 (Bytecode)”数据结构。比特码本身是一种对机器码形式的抽象，它的信息密度更高，因此相比于基线编译器生成的未经优化的机器码，**Ignition** 解释器生成比特码的速度更快，同时这些比特码的体积更小，占用的堆内存也更少。这些比特码一部分会被 **Ignition** 自身直接高效地解释和执行，另一部分则会被送往 TurboFan 的“图”生成器中等待进一步的优化。

与之前类似，如果优化编译器 TurboFan 生成的优化机器码不可用，即 V8 引擎所做出的优化假设是不成立的，那么这些机器码便会被直接丢弃，同时整个编译流程会直接再次返回到 **Ignition** 解释器，由 **Ignition** 来处理和执行这些 JavaScript 源代码。这种新的编译器链路使得 V8 引擎的整体架构复杂度大幅下降，仅有一次的 Parsing 过程也使得 JavaScript 代码可以被更高效地运行。同时也解决了在老版本链路中基线编译器会耗费大量堆内存的问题。

但实际上，从老版本 V8 链路到全新链路架构的升级并不是一蹴而就的。因此，在 Chrome 53 版本中采用的 V8 编译器链路仍旧如图 1-13 所示。由于 TurboFan 优化编译器本身的处理性能并不足以独立支撑整个 V8 链路对 JavaScript 代码的优化，因此我们不得不把 Crankshaft 优化编译器重新加回到链路中。而另一方面，由于 Crankshaft 本身没有可用于处理比特码的编译器前端，因此从 Crankshaft 进行去优化过程时仍需要把这部分代码重新交回到 Full-codegen 基线编译器的手上，所以 Full-codegen 也被重新加回到了 V8 引擎链路中。

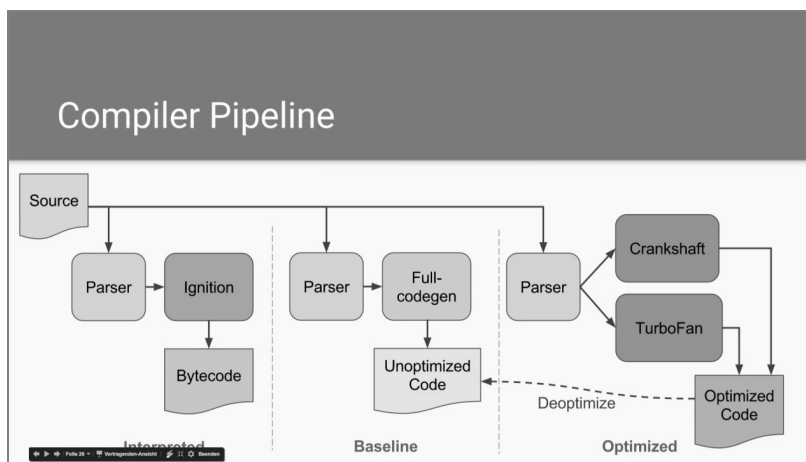


图1-13 Chrome 53版本中实际使用的Chrome V8编译器链路

可以看到，从老版本 V8 链路到全新链路架构的升级过程并不是能够快速完成的。随着 Web 应用的规模不断增大，V8 引擎需要不断地进行升级来提升自己处理 JavaScript 代码的性能。但是，每一次升级所花费的时间和 Web 应用逐渐复杂化的时间周期并不成正比。并且 V8 引擎所存在的这些问题并不是其独有的，包括 SpiderMonkey 和 JavaScriptCore 等在内的这些常见的 JavaScript 引擎均存在类似的问题。

1.2 曾经尝试——ASM.js 与 PNaCl

虽然对诸如 V8 与 JavaScriptCore 等 JavaScript 执行引擎的性能优化过程并没有想象中那么简单和快速，但人们对 Web 应用进行性能优化的尝试却从未停止过。诸如 ASM.js 与 PNaCl 等新技术的出现，为构建高性能的基于浏览器的前端 Web 应用提供了多种新的可能。

1.2.1 失落的 ASM.js

上一节我们曾介绍过，由于 JavaScript 本身是一种弱类型的编程语言，因此对于代码中某个语法环境上下文内的变量，只有当 V8 引擎解析和执行到该处的代码时才能推断出这些变量存储的具体数据类型。不仅如此，为了能够降低 JavaScript 引擎在进行变量类型推断时带来的成本开销，基于 JIT 的优化编译器通常会使用“类型特化”这种方法来优化代码的执行。

比如，对于一段包含有大量循环逻辑的 JavaScript 源代码，基线编译器首先会将这段代码标记为“warm/hot”即热代码，随后这段代码会被送往优化编译器进行优化。而在优化的过程

中，基线编译器所做出的假设就包含了“类型特化”的一部分工作。基线编译器会假设在循环内部的所有次迭代中，这段代码运行时上下文环境内各变量的类型都不会发生改变。因此当优化编译器在优化这段 JavaScript 代码时，不会再为循环体后面若干次迭代中的变量进行类型推断了，这样便部分提高了代码的整体执行效率。

可见，如果我們可以在 JavaScript 代码运行之前便告知编译器源代码中各个变量的具体类型，同时保证这些变量的类型不会在程序运行的过程中发生改变，那么编译器在编译 JavaScript 代码时便可以完全省去类型推断的过程，甚至优化编译器也可以省略去优化的过程，整个 Web 应用的运行效率将会有着巨大的提升。而 ASM.js 便是为此而生的。

“ASM.js 标准始于 2013 年 8 月，它是 JavaScript 的一个严格子集，是一种可用于编译器的低层级的、高效的目标语言。该子语言有效地为内存不安全语言（如 C 或 C++）描述了一个沙盒虚拟机的运行环境。一种静态和动态验证相结合的方式使得 JavaScript 引擎能够使用 AOT（Ahead-Of-Time，静态编译）的优化编译策略来验证 ASM.js 的代码。”这是官方给出的对 ASM.js 这项技术的标准简介。乍一看上去很抽象，那么我们就先从最直观的部分开始讲起。下面给出的是一段标准的 ASM.js 模块代码，让我们先从代码的层面来尝试了解 ASM.js 的独特面貌。

```
var asm = (function(stdlib, foreign, heap) {  
    'use asm';  
    ...  
    function __Z3addii($0, $1) {  
        $0 = $0|0; // $0 变量被声明为整型  
        $1 = $1|0; // $1 变量被声明为整型  
        var $2 = 0, label = 0, sp = 0;  
        sp = 0;  
        $2 = (($1) + ($0))|0;  
        return ($2|0); // $2 变量被声明为整型  
    }  
    ...  
    return {  
        add: __Z3addii  
    }  
})(window, null, new ArrayBuffer(0x10000));
```

类型 Annotation

从上面给出的 ASM.js 代码中可以看到，ASM.js 代码的本质其实还是 JavaScript 代码。ASM.js 并不是一种新的编程语言，它是 JavaScript 的一个严格子集。相较 JavaScript 而言，ASM.js 使

用了一种名为“Annotation（注解）”的变量类型声明方式来与 JavaScript 引擎形成对代码中使用到的变量类型的约定。JavaScript 引擎在解析 ASM.js 代码时，会根据之前所做出的约定来判断对应变量的类型，从而省去了类型推断的过程。不仅如此，如 V8 引擎链路上的 TurboFan 等优化编译器，它们在处理 ASM.js 代码时，甚至会对这些代码进行静态编译（AOT），进而让代码更加高效地执行。

比如上述代码中的赋值语句“\$0 = \$0|0”便是一种 ASM.js 的 Annotation 类型声明方式，通过这种对变量使用“按位或”运算符并与数字 0 进行“按位或”操作的声明方式，可以让 JavaScript 引擎在解析这段 ASM.js 代码时强制将该变量（\$0）视为一个 32 位（bit）的整数，并且该变量所能够存储的数据类型在程序后续的运行过程中也是无法被更改的。除了声明 32 位（bit）的整型变量，ASM.js 标准中还提供了对 double 和 float 两种数据类型的 Annotation 声明方式，详细的注解方式参考如下。

```
x:Identifier = x:Identifier|0; // 32 位整型
x:Identifier = +x:Identifier; // 双精度浮点类型
x:Identifier = f:Identifier(x:Identifier); // 32 位单精度浮点类型
```

在 ASM.js 的语法规则中，不仅需要对函数所传入的参数进行类型声明，包括函数中用到的变量、函数的参数，以及函数中 return 语句返回的结果值都需要通过 Annotation 方式进行相应的类型声明，而且这些是必需且强制的。

模块的结构

ASM.js 一般通过模块来组织自己的代码，下面给出的是一个标准 ASM.js 模块的基本结构。通过这种模块化的结构，ASM.js 可以保证模块内部的所有代码都遵循自己独有的标准和语法规则，即所有模块内部使用到的变量都保证已经通过 Annotation 的方式进行了强制类型声明。除此之外，ASM.js 模块作为一个整体也在代码层面与原始的 JavaScript 代码进行了隔离，同时其内部还可以通过暴露出的接口来与标准 JavaScript 代码进行交互。

```
function MyAsmModule (stdlib, foreign, heap) {
    "use asm";
    // 变量定义
    var variable = 0;
    // 函数定义
    function bodyFunction () { ... }
    // 函数导出
    return {
        export1: fl,
```

```
    export2: f2,  
    // ...  
};  
}  
  
var heap = new ArrayBuffer(0x10000);  
var asmModule = MyAsmModule(window, null, heap);
```

从整体上看，ASM.js 模块是一个标准的 JavaScript 函数。在这个函数内部的第一行，我们需要使用“use asm”这种标记来对模块进行声明。通过该标记，JavaScript 引擎能够得知该函数内部的所有代码都遵循着 ASM.js 的语法和规则，即整个函数代表了一个标准的 ASM.js 模块。一个完整的 ASM.js 模块其内部被分为三个部分：变量定义、函数定义和函数导出。模块导出函数可以被其他 ASM.js 模块引用，或者直接在 JavaScript 环境下通过 JavaScript 代码来调用运行。整个 ASM.js 模块在初始化时需要传入如下三个必备的参数。

stdlib

该参数包含了对一组 JavaScript 内置原生标准库的引用。由于 ASM.js 的应用场景大多数是需要进行高效处理的复杂数学计算，因此在这里一般传入的是 window 全局对象。我们可以在 ASM.js 模块内定义的函数中使用这些标准库提供的方法，比如 window.Math 对应的“JavaScript 数学运算标准库对象”就会经常使用到。ASM.js 在其标准中严格规定了一些可以在模块定义中使用的 JavaScript 标准库函数，如果在定义 ASM.js 模块方法时使用到这些标准库函数，则 JS 引擎在编译代码时会自动使用这些函数所对应的已经静态编译好的 AOT 版本代码，从而在最大程度上提高代码的执行效率。这些标准库函数如图 1-14 所示。

foreign

该参数包含了对模块外部一些自定义 JavaScript 方法的引用。通过该参数，我们可以在 ASM.js 模块中调用模块外部自定义的 JavaScript 方法，比如一些负责处理 DOM 对象或操作 Canvas 的方法。ASM.js 本身无法处理的各类场景都可以放到模块外部的函数中进行统一处理，然后再通过该参数在模块内部进行引用。如果不需要使用外部自定义的 JavaScript 方法，则可以将该参数置为 null。

heap

该参数保存着对当前 ASM.js 模块堆内存的引用，所有 ASM.js 模块外部的数据都需要被存放到该堆内存中才能够在 ASM.js 模块内部使用。同时，在 ASM.js 模块内部也只能通过该参数来获取模块外部的数据。

Standard Library	Type
Infinity NaN	<u>double</u>
Math.acos Math.asin Math.atan Math.cos Math.sin Math.tan Math.exp Math.log	(<u>double?</u>) → <u>double</u>
Math.ceil Math.floor Math.sqrt	(<u>double?</u>) → <u>double</u> ^ (<u>float?</u>) → <u>float</u>
Math.abs	(<u>signed</u>) → <u>signed</u> ^ (<u>double?</u>) → <u>double</u> ^ (<u>float?</u>) → <u>float</u>
Math.min Math.max	(<u>int</u> , <u>int</u> ...) → <u>signed</u> ^ (<u>double</u> , <u>double</u> ...) → <u>double</u>
Math.atan2 Math.pow	(<u>double?</u> , <u>double?</u>) → <u>double</u>
Math.imul	(<u>int</u> , <u>int</u>) → <u>signed</u>
Math.fround	fround
Math.E Math.LN10 Math.LN2 Math.LOG2E Math.LOG10E Math.PI Math.SQRT1_2 Math.SQRT2	<u>double</u>

图1-14 ASM.js模块严格可用的JavaScript数学标准库函数

之所以要为 ASM.js 模块设置独立的运行时堆内存，是由于标准的 ASM.js 模块在被 JavaScript 引擎解析和执行之前便会经过 AOT 处理被编译成静态代码，这部分静态代码有着固定的可用内存段。模块运行时对变量的内存分配也统一在这个固定大小的堆内存上进行。因此，正是由于 JavaScript 和 ASM.js 的运行时差异，导致我们只能通过这个独立于 JavaScript 运行时环境的堆内存段来向 ASM.js 模块传递外部数据。

与普通的 JavaScript 代码相比，ASM.js 的大部分性能提升都得益于变量严格的类型一致性，并且几乎没有垃圾回收（Garbage Collection, GC）的过程。包含数据的堆内存段需要被手动管理。相对于 JavaScript 的运行时模型而言，ASM.js 的模型更加简单——没有动态化的行为，没有内存分配和释放，只是一组预先定义好的整数和浮点数操作流程，这样的简化模型可以被深度优化，进而高效地执行。

本质上, ASM.js 的堆内存是以 `ArrayBuffer` 的形式存在的。通过初始化 ASM.js 模块时传入的三个参数, 我们可以在模块内部使用到外部自定义的 JavaScript 方法和数据。ASM.js 模块的运行时环境和原生 JavaScript 代码的运行时环境两者之间通过堆内存以 `ArrayBuffer` (`ArrayBuffer` 对象被用来表示一个通用的具有固定长度的原始二进制数据缓冲区, 即一块大小固定的内存区域) 的形式进行数据共享。下面是官方给出的一个完整的 ASM.js 模块示例, 源代码如下。

```
const START = 0;
const END = 10;

// ASM.js 模块主体
function GeometricMean(stdlib, foreign, buffer) {
  // 使用该标记声明模块类型
  "use asm";
  // 这里用临时变量存储 JavaScript 标准库中的方法
  var exp = stdlib.Math.exp;
  var log = stdlib.Math.log;

  // 从共享堆内存中获取从 JavaScript 环境传递给模块的数据
  // 这里通过 TypedArray 对象从 ArrayBuffer 中读取特定的数据类型
  var values = new stdlib.Float64Array(buffer);

  function logSum(start, end) {
    start = start|0;
    end = end|0;

    var sum = 0.0, p = 0, q = 0;

    // 这里通过将循环指针左移三位来获得对应指针在字节数组中的索引位置
    for (p = start << 3, q = end << 3; (p|0) < (q|0); p = (p + 8)|0) {
      sum = sum + +log(values[p >> 3]);
    }

    return +sum;
  }

  // 定义用于计算几何平均数的方法
  function geometricMean(start, end) {
    // 声明参数类型为整型
    start = start|0;
```

```
    end = end|0;

    return +exp(+logSum(start, end) / +((end - start)|0));
}

return { geometricMean: geometricMean };
}

function init (buffer, start, end) {
    // 使用 DataView 对象从 ArrayBuffer 中读写数据
    var dv = new DataView(buffer);
    for (let i = start; i < end; i ++) {
        // 每一次偏移 8 字节，使用小字节序来存储数据
        // 小字节序：低位字节排放在内存低地址端，高位字节排放在内存高地址端
        dv.setFloat64(i * 8, i + 1, true);
    }
}

// 初始化一块长度固定的内存用于共享数据，大小为 64KB
var heap = new ArrayBuffer(0x10000);

// 从 JavaScript 环境填充数据到共享堆内存段
init(heap, START, END);

// 初始化 ASM.js 模块并导出其内部的方法
var fast = GeometricMean(window, null, heap);

// 调用 ASM.js 内部的方法并计算结果
fast.geometricMean(START, END);
```

在这个例子中，我们构建了一个可以用于计算几何平均数的 **ASM.js** 模块，通过给出的序列首尾值（常量 **START** 和 **END**），使用该模块来计算该首尾值之间对应步长为 1 组成的序列中所有元素的几何平均数。

首先来看模块的内部结构。根据 **ASM.js** 标准中对模块定义的语法规则，在模块内部第一行需要使用“**use asm**”标记来对模块进行声明，JS 引擎会通过检验该标记来判断当前正在编译的 **JavaScript** 方法是否是一个标准的 **ASM.js** 模块。接下来，我们在模块的开头部分定义了一些变量用于引用 **JavaScript** 标准库中的方法，同时也从模块的共享堆内存中取出了从外部 **JavaScript** 环境传入的数据。紧接着是模块内部的自定义方法，这些方法承担了模块最主要的运算逻辑和

对外开放的业务功能。在模块定义的最后部分，我们将其内部的 `geometricMean` 方法作为模块的对外接口进行导出，以供外部 JavaScript 环境使用。

在初始化该 ASM.js 模块时，需要将模块用到的数据通过共享的堆内存间接地传递给模块内部的业务处理函数。我们之前介绍过，ASM.js 模块的堆内存本质上就是通过 JavaScript 的原生 `ArrayBuffer` 对象来实现的，因此这里需要将模块中使用到的数据直接存储到这个 `ArrayBuffer` 对象所对应的内存块中。首先，我们通过 `new` 关键字手动创建一个大小为 64KB 的 `ArrayBuffer` 对象。由于 `ArrayBuffer` 对象本身仅表示一个大小固定的内存块（类似于 C 语言中通过 `malloc` 函数分配的大小固定的内存段，可以把创建出来的 `ArrayBuffer` 对象理解成一个指向固定大小内存段的指针），我们不能像操作数组一样直接通过索引值来操作 `ArrayBuffer` 对象。因此，这里需要使用 ES 5 标准提供的 `DataView` 对象来与 `ArrayBuffer` 对象进行交互。`DataView` 对象提供了一个与平台内存字节序无关的底层接口，可以对 `ArrayBuffer` 对象进行多数据类型的读写操作。

通过使用 `DataView` 对象，我们向 `ArrayBuffer` 对象对应的内存段以小字节序的方式写入了从 1 到 10 共 10 个整数，并且每个整数各占据 8 字节的内存空间。接下来，通过调用 ASM.js 模块的构造方法对其进行初始化，在初始化时需要将 ASM.js 模块用到的三个参数依次传入其构造方法中。在 ASM.js 模块内部定义的方法中，我们需要使用 `TypedArray` 从共享的 `ArrayBuffer` 堆内存中读取所需的数据。从代码中可以看到，模块内部的 `logSum` 方法直接从传入的用于初始化模块的共享堆内存中读取之前写入的数据（数字 1 到 10）。在代码的最后，我们调用 ASM.js 模块对外暴露出的 `geometricMean` 方法，在该方法的内部会计算出数字 1 到 10 共 10 个数字的几何平均数，然后将最终计算结果返回。

C/C++与 Emscripten 工具链

实际上，如果需要在实际的业务项目中使用 ASM.js 这项技术，我们并不会或很少会根据业务需求直接编写 ASM.js 代码。为了能够让 JavaScript 引擎尽可能地对 ASM.js 模块代码进行优化，我们需要严格按照 ASM.js 官方提供的语法标准来组织 ASM.js 代码，这在某种程度上也大大提高了该技术的接受和学习成本。ASM.js 在诞生之初是作为一种浏览器端的底层的、高效的编译器目标语言而存在的，ASM.js 技术为那些基于强类型的静态语言提供了在 Web 端可用的虚拟机抽象实现。换句话说，ASM.js 为那些用强类型静态语言（比如 C/C++）编写的应用程序提供了一种可以直接无痛地跨平台到 Web 端运行的可能。而在实现跨平台运行的同时，也最大程度地保留了原生应用所具有的高性能特性。

比如对于下面给出的 C++ 应用代码，我们可以通过 Emscripten 工具链将其直接转译成符合 ASM.js 语法规则的 JavaScript 代码。Emscripten 是一个基于 LLVM 构建的开源项目，通过它我

们可以将 C/C++ 应用程序的源代码直接转译成经过深度优化的 ASM.js 代码, 使得这些原本需要静态编译的 C/C++ 代码可以直接在浏览器中高效地执行。Emscripten 工具链主要由两部分组成: 一个用于从源语言生成 LLVM 中间代码的编译器前端 (Emscripten Compiler Frontend); 以及一个用于将 LLVM 编译到目标语言的编译器后端 (Fastcomp)。关于 Emscripten 工具链和 LLVM 更深入的细节, 我们会在后面的章节中进行介绍。

```
bubble_sort.cc
// 引入需要的头文件
#include <emscripten/emscripten.h>
#include <iostream>
#include <vector>

using namespace std;

#ifdef __cplusplus
extern "C" {
#endif

    // 用于打印格式化数据的工具函数
    void print(const vector<int>& L) {
        for (auto i : L) {
            cout << i << " ";
        }
        cout << endl;
    }

    // 冒泡排序函数
    void bubbleSort(vector<int>& L) {
        size_t len = L.size();
        int temp;
        for (size_t i = 0; i < len; i++) {
            for (size_t j = 0; j < len - i - 1; ++j) {
                if (L[j+1] < L[j]) {
                    temp = L[j];
                    L[j] = L[j+1];
                    L[j+1] = temp;
                }
            }
        }
    }
}
```



```
// 主函数
int main () {
    vector<int> L = {49, 38, 65, 97, 76, 13, 27, 49, 55, 4};
    bubbleSort(L);
    cout << "Sorting result: ";
    // 打印排序结果
    print(L);

    return 0;
}
#ifdef __cplusplus
}
#endif
```

上面这段标准的 C++ 代码的主要功能是对一个给定向量（Vector）容器内的 10 个整数进行冒泡排序，并在排序之后将向量中的最终排序结果打印出来。为了能够让这段 C++ 程序“无痛”地运行在 Web 浏览器端，我们可以通过如下命令，借助 Emscripten 将 C++ 代码转译为与其相对应的 ASM.js 代码。

```
sudo emcc -O3 -s ASM_JS=1 -std=c++11 bubble_sort.cc
```

当 Emscripten 工具链将对应的 C++ 代码转译成 ASM.js 代码后，会在当前目录下自动生成一个名为“a.out.js”且含有 JavaScript 代码的文件。在这个文件中，Emscripten 工具链会自动配置好所有 ASM.js 模块初始化和运行前需要设置的环境变量，以及构造函数需要传入的参数，比如 ASM.js 模块与 JavaScript 运行时环境共享的堆内存段等。接下来，我们便可以在命令行中直接通过 Node.js 命令来运行这个含有 ASM.js 模块的 JavaScript 脚本文件。

```
node -trace_asm_parser -validate_asm a.out.js
// [asm.js translation successful: time=25.440ms, translate_zone=1906KB, compile_zone+=671KB]
// Sorting result: 4 13 27 38 49 49 55 65 76 97
```

ASM.js 性能测试

至此，我们已经介绍了 ASM.js 的基本原理，以及相较于原生 JavaScript 代码，ASM.js 作为其严格子集在性能方面所体现出来的优势。接下来，我们会分别使用 ASM.js、原生 JavaScript 和 C++ 这三种语言（代码）来编写具有同样业务逻辑的一个简单应用，并分别在其各自不同的环境中运行，借此通过对比来了解 ASM.js 在性能优化上能够达到的程度。这里我们将要编写一个可用于计算斐波那契数列的应用程序（函数）。

首先给出的是基于原生 JavaScript 编写的应用源代码。

```
function fib(x) {  
  if (x < 2) {  
    return 1;  
  } else {  
    return fib(x - 1) + fib(x - 2);  
  }  
}  
  
// 使用 Performance API 来测量时间会更加稳定和精准  
let startTime = performance.now()  
console.log(fib(45));  
console.log(`${performance.now() - startTime} ms`);
```

接下来给出的是基于 C++编写的应用程序源代码。

```
#include <iostream>  
#include <ctime>  
  
using namespace std;  
int fib (int x);  
  
int fib (int x) {  
  if (x < 2) {  
    return 1;  
  } else {  
    return fib(x - 1) + fib(x - 2);  
  }  
}  
  
int main () {  
  // 通过计算 CPU 时钟经过的周期数来计算函数调用花费的时间 (也可以使用 chrono 提供的 C++标准时间库)  
  clock_t start = clock();  
  printf("%d\n", fib(45));  
  clock_t end = clock();  
  
  // 将得到的 CPU 时钟周期数换算成时间  
  printf("Time Cost: %lums\n", (end - start) * 1000 / CLOCKS_PER_SEC);  
  
  return 0;  
}
```

最后给出的是基于 ASM.js 实现的应用程序源代码。这里的 ASM.js 应用程序源代码是直接

通过 Emscripten 工具链从上述 C++ 应用程序源代码转译而来的。由于 Emscripten 在将 C++ 源代码转译成 ASM.js 代码时会对代码本身进行各种各样的优化,因此在生成代码中存在的变量大多并不具有较强的可读性。这里我们只列出了该 ASM.js 模块最核心的数据处理和运算逻辑部分的代码,如下所示。

```
...
var asm = (function(global, env, buffer){
    'use asm';
    ...
    function __Z3fibi($0) {
        $0 = $0|0;
        var $1 = 0, $10 = 0, $11 = 0, $12 = 0, $2 = 0, $3 = 0, $4 = 0, $5 = 0, $6 = 0, $7 = 0,
        $8 = 0, $9 = 0, label = 0, sp = 0;
        sp = STACKTOP;
        STACKTOP = STACKTOP + 16|0; if ((STACKTOP|0) >= (STACK_MAX|0)) abortStackOverflow(16|0);
        $2 = $0;
        $3 = $2;
        $4 = ($3|0)<(2);
        if ($4) {
            $1 = 1;
            $12 = $1;
            STACKTOP = sp;return ($12|0);
        } else {
            $5 = $2;
            $6 = (($5) - 1)|0;
            $7 = (__Z3fibi($6)|0);
            $8 = $2;
            $9 = (($8) - 2)|0;
            $10 = (__Z3fibi($9)|0);
            $11 = (($7) + ($10))|0;
            $1 = $11;
            $12 = $1;
            STACKTOP = sp;return ($12|0);
        }
        return (0)|0;
    }
    ...
})(Module.asmGlobalArg, Module.asmLibraryArg, buffer);
...
```

最后我们将这三种基于不同语言（代码）编写的对应于同一业务功能的应用其各自的运行结果用柱状图统计出来，如图 1-15 所示。可以看到，相较于经过优化（命令行中的“-O3”参数指定了编译器需要对代码进行的优化程度）的 C++代码生成的应用程序，基于 ASM.js 的代码其执行效率还是低了一些。但另一方面，相较于使用原生 JavaScript 编写的应用，JS 引擎内部的优化编译器对 ASM.js 代码做出的性能优化可以说还是十分显著的。（评测结果仅供参考）

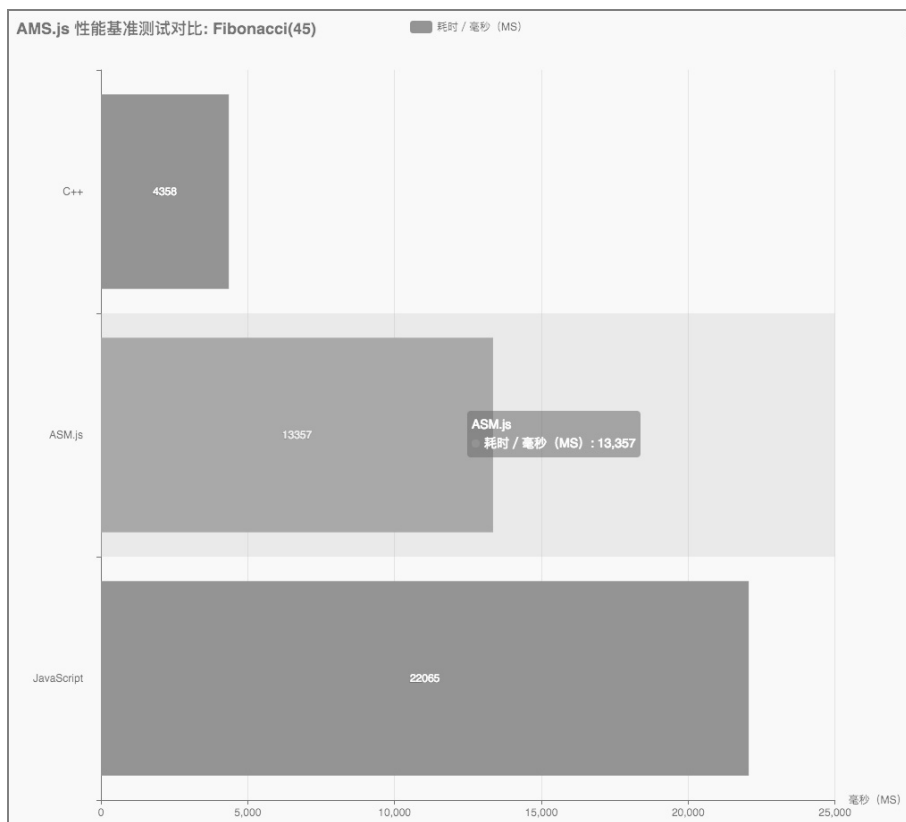


图1-15 ASM.js性能基准测试对比

应用领域

ASM.js 标准自 2014 年发布至今已经鲜有人关注了。但在 ASM.js 标准刚刚发布的早期阶段，很多知名的互联网公司和厂商都在不断尝试将 ASM.js 这项技术应用在自己公司的产品上。知名的游戏开发引擎厂商 Unity Technologies 于 2015 年将 ASM.js 技术集成到自家的 Unity3D 游戏开发引擎中，这种改变使得通过 Unity3D 引擎开发出来的大型主机游戏可以被高效且相对流畅地运行在 Web 浏览器上。

借助 Emscripten 工具链的能力，那些基于 Unity3D 开发的使用了 OpenGL（Open Graphics Library）技术的游戏和应用，都可以被快速无痛地移植到支持 ASM.js 技术的 Web 平台上。在 Web 端运行这些游戏和应用时，浏览器会自动使用 WebGL 来替代原有的 OpenGL（OpenGL 和 WebGL 均基于 OpenGL 标准实现，只是针对不同的平台），因此其功能和性能均得以保留和兼顾。

Unity3D 引擎使用了一套名为“IL2CPP”的脚本运行时环境，为那些基于其开发的游戏和应用提供了跨平台的可能。IL2CPP 运行时环境的整体结构如图 1-16 所示，位于 Unity3D 用户端的 Mono 编译器会将 Unity 开发中最上层的基于 C# 语言编写的开发脚本直接编译成 IL（Intermediate Language）的形式。这里的 IL 主要指 MSIL（微软中间语言）。MSIL 是一种在 .NET 程序被编译成机器语言的过程中产生的中间表示语言。该语言本身是独立于具体的 CPU 架构的，但同时又可以有效地被编译器直接转换为与架构相关的机器语言。接下来，一个名为“il2cpp.exe”的 AOT 编译器会将之前生成的 MSIL 直接编译成对应的 C++ 代码。然后，这些 C++ 代码又会被分别编译到包括浏览器（以 ASM.js 的形式存在）在内的各类不同的运行平台上。除此之外，IL2CPP 本身还提供了一个用于运行时环境的脚本库，这个脚本库被用来支持 C++ 的虚拟机环境。在这个脚本库中提供了一些与平台无关的服务和特性如垃圾回收和多线程等，可以让那些使用到这些特性的 C++ 代码被无痛地移植到这些平台上。

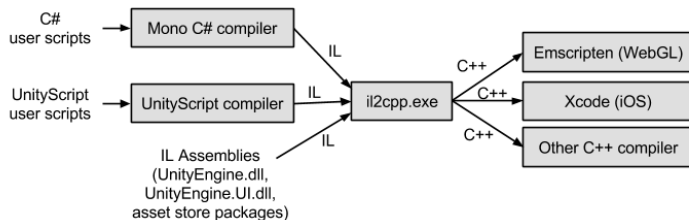


图1-16 IL2CPP运行时环境的整体结构

除 Unity3D 引擎之外，另一个知名的游戏引擎设计厂商 Epic Games 也早在 2013 年便宣布，其旗下的 Unreal Engine3 游戏开发引擎开始支持创建基于 ASM.js 技术的 Web 端跨平台游戏。不仅仅是在游戏开发引擎领域，基于 ASM.js 实现的可用于前端的 SQLite 数据库、ZLib 压缩算法库，甚至是可以直接用于 Web 浏览器的 VIM 编辑器，它们大多都是在 ASM.js 技术的诞生初期，从其原先对应的 C/C++ 项目通过 Emscripten 工具链转译成 ASM.js 项目的。

向下兼容

由于 ASM.js 本身只是 JavaScript 的一个严格子集，因此，对于一些不支持 ASM.js 格式或无法识别 ASM.js 模块并对其进行优化的 Web 浏览器来说，仍然可以正常执行这些基于 ASM.js 编写的应用代码，只不过浏览器只能将这些 ASM.js 代码当作原生的 JavaScript 代码来进行处理。

包括 AOT 在内的一些专门用于 ASM.js 代码的优化手段并不会被执行，因此代码的整体执行效率也会大打折扣。

低谷

ASM.js 技术本身也存在着一定的问题和局限性。比如在 ASM.js 标准中，只定义了对数值类型变量的 Annotation 声明方式，这使得 ASM.js 的应用场景大部分都集中在对基于 Web 浏览器端的数值计算密集型应用的优化处理上。而对于其他常见的如字符串和对象等 JavaScript 类型，我们只能通过比如将这些数据编码成内存中一段连续的 32 位整数等方式来供 ASM.js 进行优化和处理。不仅如此，各大浏览器厂商对 ASM.js 标准的支持程度和实现方式也不尽相同。同样的一段 ASM.js 代码在各个浏览器上的运行效果可能会有很大的不同，而这也成为阻碍 ASM.js 发展的一个关键性因素。

Mozilla 作为一家最早支持 ASM.js 特性的浏览器厂商，自 2013 年 2 月起便着手开发在 Firefox 上为 SpiderMonkey 引擎准备的用于支持 ASM.js 特性的优化编译器 OdinMonkey。虽然 OdinMonkey 编译器已在 Firefox 22 版本中发布，但其本身遗留下来的问题却一直都没有得到解决。很多已经被报告出来的开放性 Bug 早已无人问津，无人进行维护。

一项新技术的出现一般是由于在业务实现上有着相对应的需求。而这项新技术是否能够持续不断地发展下去，也取决于在实际的业务应用中，该技术对业务本身是否起到了正向的积极作用。ASM.js 之所以没能被持续地发展下去，也正是由这项技术本身所存在的一些问题导致的，比如其应用场景较少、各主流浏览器对标准的实现不统一，以及使用成本较高等。

ASM.js 标准自 2014 年 8 月发布的最后一次更新日志后，至今已经没有了进一步的消息。

1.2.2 古老的 NaCl 与 PNaCl

NaCl 是谷歌 Chrome 开发团队于 2011 年 8 月在 Chrome 14 版本浏览器中发布的一项新技术。通过该技术，我们可以让基于 C/C++ 语言编写的应用程序安全、高效地运行在 Web 浏览器端，并且不依赖用户所使用的具体操作系统类型。NaCl 的全称是“Google Native Client”，其暗示了谷歌想要把基于原生 C/C++ 语言编写的本地应用程序运行在 Web 浏览器上的决心。谷歌官方声称其开发 NaCl 这项技术的目的是为了能够在保障安全性和可移植性的前提下，尽量缩小桌面原生应用与 Web 应用之间的界限，这使得我们可以将一批传统的桌面原生应用无痛地移植到 Web 平台上来运行。伴随 NaCl 技术同期出现的还有 Google 自研的，基于 Chrome 浏览器核心开发的桌面操作系统——Chrome OS。

基于 NaCl 技术开发出来的应用可以以接近原生 C/C++ 应用的效率在浏览器端稳定地运行。同时, NaCl 应用也可以直接使用 CPU 暴露出的包括 SIMD (单指令多数据流) 和基于共享内存的多核并行处理在内的所有高级特性。为了能够让 NaCl 应用安全且高效地运行在 Web 浏览器端, Chrome 使用了一种名为“SFI (Software Fault Isolation, 软件故障隔离)”的技术来让 NaCl 应用运行在浏览器的独立沙盒环境中。

可以直接操作内存是 C/C++ 语言本身所独有的特性, 因此可以通过 C/C++ 指针来访问应用程序运行时所处的内存段, 这使得我们可以完全自主地去管理应用程序运行时的内存分配, 从某种程度上讲, 借助 C++ 中诸如“移动语义”等高级特性, 可以大幅度提升应用程序的整体运行效率。但另一方面, 也正是由于 C/C++ 语言这种可以操作系统底层资源的能力, 使得某些基于 NaCl 技术跨平台运行在浏览器上的具有非法 C/C++ 代码的原生应用可能会对浏览器本身甚至用户产生危害。借助 SFI 技术, 浏览器可以让这些 C/C++ 应用运行在一个内存独立的沙盒环境中。沙盒环境提供了一个可供 C/C++ 应用使用的运行时环境。总的来讲, SFI 技术的作用就是为了保证那些安全性未知的代码不会在运行过程中对操作系统本身产生危害。

比如在图 1-17 所示的代码中, 原始 C/C++ 代码的主要功能是对指针 `p` 所指向的内存段填充数据, 并且在每次填充完数据后, 指针 `p` 都会向内存高位继续移动。但事实上, 把所有的内存地址段都暴露给应用程序, 让其可以任意地对内存中的数据读写操作, 这种行为是十分危险的。比如, 如果应用程序改写了用于存储浏览器内核配置信息的内存地址段中的数据, 这势必会造成浏览器的功能异常甚至崩溃。而对于一个含有用户敏感信息的内存地址段, 允许对其进行任意读写操作也势必会造成用户隐私的泄露。通过 SFI 技术, 浏览器会对这些运行在 Web 端的 C/C++ 代码进行安全性改写, 比如为与内存读写操作相关代码中的指针设置可用的边界值, 一旦指针引用越过边界值, 便会抛出错误, 同时停止应用的运行。当然, 这只是 SFI 技术为我们提供的众多安全保障中的“冰山一角”。

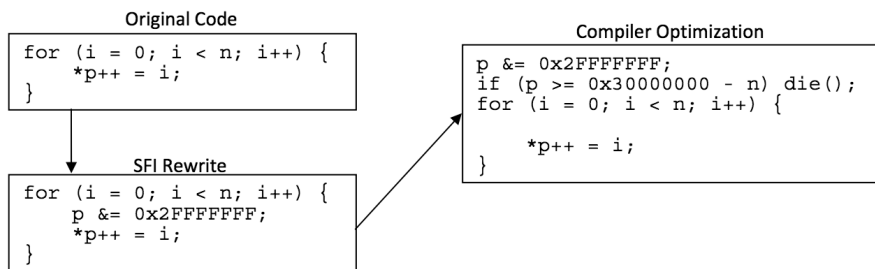


图1-17 SFI重写过程示例

NaCl 应用基本结构

如图 1-18 所示，一个标准的 NaCl 应用由用于构建页面样式的 HTML 标签和 CSS 样式、用于控制交互逻辑与连接 NaCl 模块的 JavaScript 代码、用于描述基于不同处理器架构编译生成的 NaCl 模块所在路径信息的 Manifest 描述文件和一个或多个使用 C/C++ 语言编写且经过编译的 NaCl 模块共四部分组成。NaCl 模块本身并不具备可移植性，在它的内部包含有针对特定处理器架构的机器码，其架构类型主要有 x86-32、x86-64、ARM 和 MIPS。

因此，为了能够让 NaCl 应用可以在基于各种处理器架构的操作系统上运行，我们需要针对各处理器架构分别单独编译其各自对应的 NaCl 模块，同时利用之前提到的 Manifest 描述文件将模块的信息进行汇总，浏览器在运行 NaCl 应用时，便会根据当前计算机的处理器架构自动从 Manifest 文件中选择合适的模块并加载到本地环境中使用。

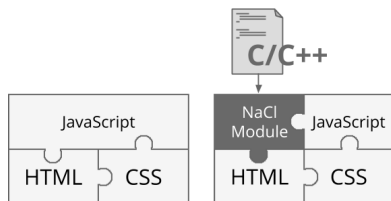


图1-18 NaCl应用的基本结构

NaCl 应用场景

借助 NaCl 技术，我们可以将现有的 C/C++ 应用无痛地跨平台到 Web 浏览器上运行。除此之外，还可以构建如视频/图片处理器等多媒体 Web 应用、基于 OpenGL ES 开发的可以运行在 Web 端的大型网络游戏、与深度学习和区块链相关的密集计算类 Web 应用，以及任何需要使用 C/C++ 来进行提速的 Web 应用等。综合来看，NaCl 的主要优势在于可以在 Web 浏览器上安全、高效地运行基于 C/C++ 语言编写的本地原生应用，同时一些与计算机底层架构相关的 CPU 特性也不会因为运行在浏览器上而被禁用。另外，结合 WebGL/OpenGL ES 提供的高效 2D/3D 视频处理性能，NaCl 在 Web 浏览器端的游戏和多媒体应用开发也会更胜一筹。

NaCl 基本原理

我们之前介绍过，NaCl 在浏览器中构建了一个独立的沙盒环境来运行原生的 C/C++ 应用。运行在沙盒环境中的 NaCl 应用会受到沙盒在可调用 API 及可访问内存段上所做的一些安全性限制。当浏览器在 HTML 标签中检测到 NaCl 模块的加载请求时（在 HTML 中可以通过 `<embed>` 标签来加载一个 NaCl 模块），浏览器会通过调用 NaCl 助手（Native Client Helper）来从远程服务器下载已经编译好的 NaCl 模块。NaCl 应用的基本调用流程如图 1-19 所示。

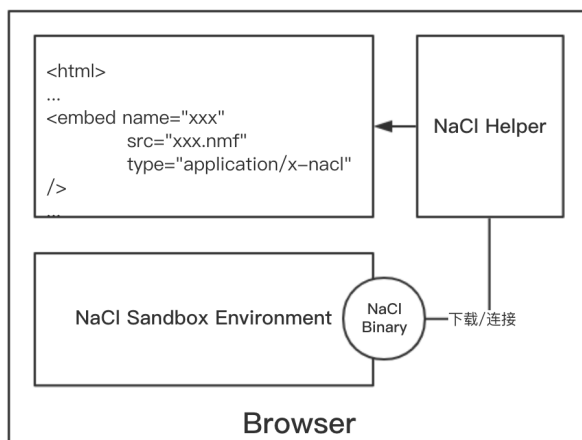


图1-19 NaCl应用的基本调用流程

在 NaCl 模块被加载和运行前，浏览器会根据 NaCl 应用需要遵守的安全规则来对该模块中的机器码进行安全检查。检查内容主要包括分析该模块是否读写了处于非安全区域的内存数据段、是否调用了受限制的 API 等。若安全检查失败，则该 NaCl 模块会被拒绝加载和运行；否则，该模块会被直接加载到沙盒环境中运行。此时，NaCl 助手会与该模块进行连接，而模块随后便可以通过 NaCl 助手与浏览器进行通信和交互。模块运行前的安全检查便是基于我们之前介绍的 SFI 技术实现的。

我们可以通过 HTML 标签 `<embed>` 以加载插件的方式来加载一个 NaCl 模块。如上面的图 1-19 所示，在 `<embed>` 标签中需要指定当前正在加载的插件其具体类型为 “application/x-nacl”，即一个标准的 NaCl 应用模块。同时，在该标签中我们还通过 `src` 属性指定了一个以 “.nmf” 为后缀的文件。在这个文件中，通过标准 JSON 格式的数据结构描述了有关该 NaCl 模块的一些信息，我们也称这个文件为 NaCl 模块的描述文件（NaCl Manifest File）。下面给出一个示例描述文件中的部分内容。

```

{
  "program": {
    "x86-64": {
      "url": "air_mech_x86_64.nexe"
    },
    "x86-32": {
      "url": "air_mech_x86_32.nexe"
    },
    "arm": {

```

```

    "url": "air_mech_arm.nexe"
  }
}
}

```

在这个 NaCl 模块对应的描述文件中，我们为 NaCl 应用指定了对应于不同处理器架构的 NaCl 模块其所在位置和名称。当浏览器解析到用于加载 NaCl 模块的 HTML 标签时，会根据其当前所处计算机的处理器架构类型从该.nmf 描述文件中找到对应 NaCl 模块所在的远程位置，并通过 HTTP 请求来加载这个模块。加载到浏览器端的 NaCl 模块随后会被 NaCl Helper 的进程（Native Client Process）进行处理。Helper 进程随后便会使用 SFI 技术对模块进行安全分析，以确保该模块中的代码调用遵循 NaCl 的安全规则。

每一个标准的 NaCl 模块都是一个以“.nexe”为后缀的 ELF 格式二进制文件，该文件可以直接在 Chrome 中加载并运行。在每一个 ELF 类型文件内容的最开始部分，都存在着一个 ELF 文件头（ELF Header），在这个文件头中记录着该文件所适用的处理器架构信息。如图 1-20 所示，我们可以通过 readelf 命令来查看一个.nexe 文件的文件头信息。从该文件头中可以得知该 NaCl 模块所适用的处理器架构、处理器位数，以及对应的 CPU 字节序类型。

```

[root@iZ23ki8vt8tZ ~]# readelf -h core_x86_32.nexe
ELF Header:
  Magic:   7f 45 4c 46 01 01 00 00 00 00 00 00 00 00 00
  Class:                           ELF32
  Data:                               2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                        0
  Type:                               EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:                0x1000a60
  Start of program headers:           52 (bytes into file)
  Start of section headers:          135996 (bytes into file)
  Flags:                               0x0
  Size of this header:                 52 (bytes)
  Size of program headers:            32 (bytes)
  Number of program headers:           9
  Size of section headers:            40 (bytes)
  Number of section headers:          29
  Section header string table index:  28

```

图1-20 .nexe文件的基本格式

由于 NaCl 模块的平台独立性，在实际项目中使用时，需要为每种不同的处理器架构分别单独编译对应版本的 NaCl 二进制模块文件。这种方式既不方便也不符合开源软件的便携特性。不仅如此，NaCl 模块的这种直接存储针对底层处理器架构机器码的方式也使得模块本身失去了可移植性。如果由于历史原因导致某一型号的处理器的架构模式已经不再被使用，那么所有对应于该处理器架构的 NaCl 模块便失去了效用。

因此，相比于将 NaCl 模块以 C/S（Client-Server）模式让开发者或用户自行存放在独立的服务器中，Chrome 官方最后规定我们只能将 NaCl 模块发布到 Chrome 网上商店以进行统一管理。也就是说，我们只能通过 Chrome Web Store 来下载和使用别人开发的 NaCl 应用，同时我们自己开发的 NaCl 应用也只能通过这个网上商店来向别人开放使用。也正是由于这个原因，另一种基于 NaCl 改进的 PNaCl 技术出现了。

PNaCl 应用基本结构

NaCl 应用在被部署到 Web 浏览器之前，我们需要根据各种类型的处理器架构来编译出各自相对应的 NaCl 二进制模块，这就导致 NaCl 应用在互联网上无法被自由地分发。PNaCl 的全称是“Chrome Portable Native Client”，相较于 NaCl 应用，PNaCl 则更加强调其便携的特性。PNaCl 并不会直接将应用的 C/C++源代码编译成依赖特定处理器架构的底层机器码，整个 PNaCl 应用的创建和运行一共分为两个步骤。首先，PNaCl 会将应用的 C/C++源代码编译成一种基于 LLVM 生成的具有抽象中间比特码格式的模块（通常以“.pexe”作为后缀），这种模块并不依赖具体的处理器架构，因此可以在互联网上被随意地分发。接下来，我们在 Web 浏览器中部署和运行该 PNaCl 应用。浏览器会首先通过 HTTP 请求将之前生成的中间比特码模块加载到内存中，然后再通过浏览器中的一个内置的 AOT 转译器根据当前用户计算机的处理器架构类型来对这个包含有中间比特码的 PNaCl 模块进行转译。转译生成的二进制文件中便包含了基于某种特定处理器架构的机器码，随后可被浏览器直接执行。

如图 1-21 所示的便是 PNaCl 与 NaCl 应用在编译时和运行时的区别。总体来说，PNaCl 是 Google 更推荐使用的方案。首先，PNaCl 与 NaCl 两者在应用的安全性、运行效率及开发方式上都基本一致，并没有较大的差异。但 PNaCl 应用不依赖具体的硬件架构，能够以更加开放的方式拥抱互联网，并且可以通过简单的 C/S 架构来进行应用的部署。PNaCl 应用对不同的底层处理器架构都具有高度可复用性，能够真正做到“一次编译，到处运行”。相比于 PNaCl 应用，NaCl 应用的这种“需要通过 Chrome 网上商店才能进行应用分发”的限制则不具备任何的软件便携性。

但值得一提的是，NaCl 在某些方面却也要优于 PNaCl。NaCl 允许在应用中使用一些依赖具体架构的底层指令代码。由于 PNaCl 的便携特性，位于运行时环境的 AOT 转译器会将这些底层指令代码替换为另一套具有同等效用的代码，而这些代码是为所有底层处理器架构设计的通用型代码，因此从整体上屏蔽了处理器架构的不同底层指令细节，但其执行效率却也有所下降。除此之外，NaCl 还支持动态链接的特性，这个是 PNaCl 所无法提供的。

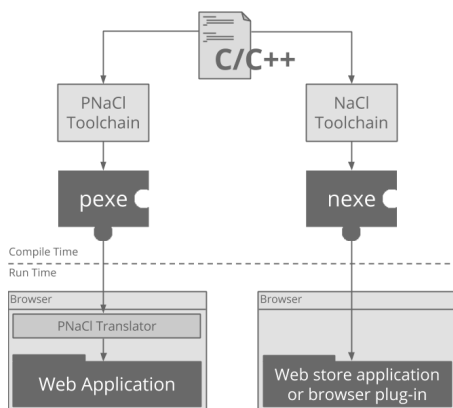


图1-21 PNaCl与NaCl应用在编译时和运行时的区别

PNaCl 实践细节

为了能够更加直观地了解一个 PNaCl 应用从创建、部署到运行的整个过程，下面我们将从一个简单的例子切入。前面我们讲到，一个完整的 PNaCl 应用包括几个部分：用于构建页面样式的 HTML 标签和 CSS 样式、用于控制逻辑与连接 PNaCl 应用的 JavaScript 代码、Manifest 模块描述文件，以及一个编译好的以“.pexe”为后缀的 PNaCl 应用模块。

首先，我们从这个核心的 PNaCl 应用模块开始入手。如图 1-22 所示，PNaCl 模块与 JavaScript 环境之间的交互都是通过一个名为“PPAPI (Pepper Plugin-in API)”的 API 接口来进行的，这里将 PPAPI 简称为“Pepper”。Pepper 允许基于 C/C++ 的 PNaCl 模块与浏览器进行通信，并且以一种安全、便携的方式来访问系统级别的功能。NaCl/PNaCl 标准中的一个十分重要的安全性约束内容是：模块不能够直接进行 OS（操作系统）级别的 API 调用，而 Pepper 便可以为模块提供已经封装好且安全可靠的用于模拟对应 OS 系统调用的一系列上层 API 接口。

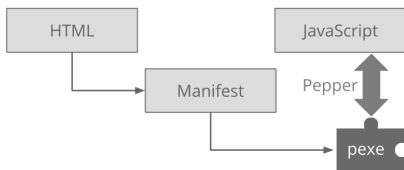


图1-22 PNaCl应用内部的基本调用逻辑

在开发 PNaCl 模块之前，请确保计算机本地已经安装并配置好了 Pepper 的本地开发 SDK 环境。这里我们还是使用 PNaCl 技术来开发一个用于计算斐波那契数列前 N 项和的简单应用。应用的具体交互流程描述如下：从 JavaScript 环境获得参数 N 并传递给该应用；应用接收到参数 N 并计算结果；最后应用将计算结果返回到 JavaScript 环境，并在控制台打印该结果。

首先我们编写用于构建该 PNaCl 模块的 C++源代码，如下所示。

```

pnacl_fibonacci.cc
// 引入 PPAPI 库头文件
#include "ppapi/cpp/instance.h"
#include "ppapi/cpp/module.h"
#include "ppapi/cpp/var.h"
#include "ppapi/cpp/var_dictionary.h"

namespace {
    // 定义将要从 PNaCl 发送到浏览器端的字符串常量
    const char* kReplyString = "Hello, this is PNaCl!";
}

class HelloTutorialInstance : public pp::Instance {
public:
    explicit HelloTutorialInstance(PP_Instance instance) : pp::Instance(instance)
    {}
    virtual ~HelloTutorialInstance() {}

    // 从浏览器端发送过来的数据首先会经过 HandleMessage 函数的处理
    virtual void HandleMessage(const pp::Var& var_message) {
        // 处理从浏览器端发送过来的数据
        if (!var_message.is_number())
            return;
        // 通过 Pepper API 处理从 JavaScript 环境传递过来的消息
        int message = var_message.AsInt();
        pp::VarDictionary var_dict;
        // 构造一个字典对象
        var_dict.Set(pp::Var("echo"), pp::Var(kReplyString));
        var_dict.Set(pp::Var("result"), pp::Var(this->fib(message)));
        // 通过 PostMessage 方法将数据返回给浏览器端
        PostMessage(var_dict);
    }
private:
    // 计算斐波那契数列的前几项和
    int fib (int x) {
        if (x < 2) {
            return 1;
        } else {

```

```

        return fib(x - 1) + fib(x - 2);
    }
}
};

// 浏览器会检查 HTML 页面中所有用于声明 PNaCl 应用的<embed>标签
// 每一个有效的标签都会生成一个与之对应的 pp::Module 对象
class HelloTutorialModule : public pp::Module {
public:
    HelloTutorialModule() : pp::Module() {}
    virtual ~HelloTutorialModule() {}
    // 模块初始化后会调用该类的 CreateInstance 方法来创建一个模块实例
    virtual pp::Instance* CreateInstance(pp_Instance instance) {
        return new HelloTutorialInstance(instance);
    }
};

namespace pp {
    // PNaCl 模块第一次加载时调用的工厂方法
    Module* CreateModule() {
        return new HelloTutorialModule();
    }
}

```

上面给出的便是对应于该 PNaCl 应用模块的全部 C++源代码，在其中我们使用到了 Pepper 提供的部分 API。模块在初始化后会接收从 JavaScript 环境传来的一个整数，然后计算该整数对应前 N 项斐波那契数列的和，最后再将计算结果返回给 JavaScript。这里 PNaCl 模块会将计算结果连同客户端需要的所有数据统一存放到一个字典数据结构中进行返回。该字典数据结构最后会被自动转换为 JavaScript 对象（Object）的形式供客户端直接使用。

一个完整的 PNaCl/NaCl 模块在 C++代码层面是由三个组件部分构成的。

- 一个名为“CreateModule”的工厂方法。PNaCl/NaCl 模块并没有类似于编写传统 C/C++ 应用时代码中名为“main”的入口方法，因此 CreateModule 便是用于连接模块与浏览器的入口方法。当一个 PNaCl/NaCl 应用被初次加载时，浏览器首先会调用该方法来初始化模块，同时该方法返回的 Module 对象会以单例模式的状态保留在内存中。
- 一个继承自 pp::Module 并实现了 CreateInstance 方法的模块类。浏览器每次遇到用于加载同一个 PNaCl/NaCl 应用模块的<embed>标签时都会调用该方法，并返回对应于该模

块的对象实例。

- 一个继承自 `pp::Instance` 并实现了 `HandleMessage` 方法的实例类。浏览器每次从 JavaScript 环境通过 `postMessage` 方法向 PNaCl/NaCl 应用发送消息时，我们都可以通过该类中定义的 `HandleMessage` 方法来获得这些消息，并在这里进行相应的业务逻辑处理。

PNaCl/NaCl 应用的底层代码调用逻辑如图 1-23 所示。

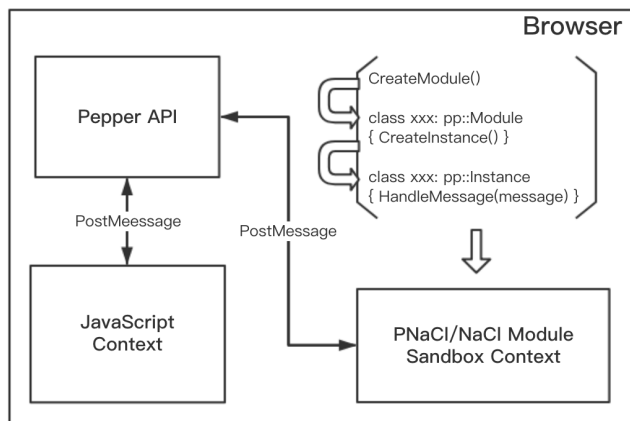


图1-23 PNaCl/NaCl应用的底层代码调用逻辑

为了能够方便地编译这段 C++ 代码，我们需要编写一个 **Makefile** 文件来帮助自动编译并生成对应的目标文件。**Makefile** 文件的内容如下。

```

#
# 获取与 Pepper SDK 相关的工具链和头文件目录
#
THIS_MAKEFILE := $(abspath $(lastword $(MAKEFILE_LIST)))
NACL_SDK_ROOT ?= $(abspath $(dir $(THIS_MAKEFILE))../..)

# 一些编译器符号
WARNINGS := -Wno-long-long -Wall -Wswitch-enum -pedantic -Werror
CXXFLAGS := -pthread -std=gnu++98 $(WARNINGS)

#
# Compute tool paths
#
GETOS := python $(NACL_SDK_ROOT)/tools/getos.py
OSHELPERS = python $(NACL_SDK_ROOT)/tools/oshelpers.py

```

```

OSNAME := $(shell $(GETOS))
RM := $(OSHELPER) rm

PNaCl_TC_PATH := $(abspath $(NACL_SDK_ROOT)/toolchain/$(OSNAME)_pnacl)
PNaCl_CXX := $(PNaCl_TC_PATH)/bin/pnacl-clang++
PNaCl_FINALIZE := $(PNaCl_TC_PATH)/bin/pnacl-finalize
CXXFLAGS := -I$(NACL_SDK_ROOT)/include
LDFLAGS := -L$(NACL_SDK_ROOT)/lib/pnacl/Release -lppapi_cpp -lppapi

# 设置编译目标
all: pnacl_fibonacci.pexe

clean:
    $(RM) pnacl_fibonacci.pexe pnacl_fibonacci.bc

pnacl_fibonacci.bc: pnacl_fibonacci.cc
    $(PNaCl_CXX) -o $@ $< -O2 $(CXXFLAGS) $(LDFLAGS)

pnacl_fibonacci.pexe: pnacl_fibonacci.bc
    $(PNaCl_FINALIZE) -o $@ $<

```

Makefile 文件编写完成后，我们将前面模块的 C++ 源代码文件和该 Makefile 文件放在同一个文件夹内，然后在命令行的当前目录下直接运行 `make` 命令，即可编译出以 “.pexe” 为后缀的目标文件，也就是 PNaCl 应用的核心模块文件。

PNaCl 模块文件创建完毕后，接下来开始编写用于控制前端页面交互和连接 PNaCl 模块的 JavaScript 代码。这里为了更加直观地展示代码内容，我们将 HTML 标签、CSS 样式和 JavaScript 代码三者写在了同一个 HTML 文件中，代码如下。

```

index.html
<!DOCTYPE html>
<html>
<head>
  <title>PNaCl Fibonacci</title>
  <script type="text/javascript">
    // 对 PNaCl 应用的引用
    let appReference = null;
    let statusText = 'NO-STATUS';
    let startTime = null;
    // 将要传递给 PNaCl 模块的整数数据

```



```
const fibonacciNumber = 45;

// Indicate load success.
function moduleDidLoad() {
    // 获取初始化标签 DOM 对象的引用
    appReference = document.getElementById('pnacl_fibonacci');
    updateStatus('SUCCESS');
    // 向 PNaCl 应用发送数据
    // 更新开始时间，在这里用来统计 PNaCl 在计算 Fibonacci 数列时花费的时间
    startTime = performance.now();
    // 发送数据
    appReference.postMessage(fibonacciNumber);
}

// JavaScript 环境的消息处理函数，当 PNaCl 向 JavaScript 发送消息时可以在这里接收消息并进行处理
function handleMessage(message_event) {
    console.log(message_event.data);
    // 计算并打印出 PNaCl 应用在计算 Fibonacci 数列时花费的时间
    let usedTime = performance.now() - startTime;
    console.log(`${usedTime} ms`);
}

// 页面载入事件
function pageDidLoad() {
    if (appReference == null) {
        // 若 PNaCl 应用的引用变量内容为空，则表示应用正在加载中
        updateStatus('LOADING...');
    } else {
        updateStatus();
    }
}

// 用来更新显示状态信息的函数
function updateStatus(opt_message) {
    if (opt_message) {
        statusText = opt_message;
    }
    var statusField = document.getElementById('statusField');
    if (statusField) {
```

```

        statusField.innerHTML = statusText;
    }
}
</script>
</head>
<body onload=pageDidLoad() ">

    <h1>NaCl C++ Tutorial: Getting Started</h1>

    <p>

        <!--

```

这里对<embed>标签的外层容器添加了两个事件监听器，一个用来监听 PNaCl 应用的加载情况；另一个用来监听从 PNaCl 应用发出的消息。可能有人会有疑问，这里为什么不直接对<embed>标签添加监听器？这是由于我们希望在 PNaCl 应用加载之前就设置好监听器，以监听应用的加载状态。由于 JavaScript 的事件冒泡和捕获特性，我们便可以在<embed>标签的外层添加监听器来捕获在该标签上发生的事件。

```

-->
<div id="listener">
    <script type="text/javascript">
        var listener = document.getElementById('listener');
        listener.addEventListener('load', moduleDidLoad, true);
        listener.addEventListener('message', handleMessage, true);
    </script>
    <!--

```

通过<embed>标签以插件的方式来加载一个 PNaCl 应用。由于该应用没有显式的图形交互，因此该 HTML 元素的可见长宽均被设置为 0。在这里我们指定了 PNaCl Manifest 描述文件的所在位置，同时指定了加载插件的类型为一个 PNaCl 应用。

```

-->
    <embed id="pnacl_fibonacci"
        width=0 height=0
        src="pnacl_fibonacci.nmf"
        type="application/x-pnacl" />
    </div>
</p>

    <h2>Status <code id="statusField">NO-STATUS</code></h2>
</body>
</html>

```

至此，我们已经编写好了 JavaScript 环境中用于处理界面交互和连接 PNaCl 模块的代码。最后一步，我们需要为这个 PNaCl 应用编写一个用于描述应用模块信息的 Manifest 文件，将两个环境中的代码进行连接。浏览器会根据该文件中的描述信息来加载对应模块并进行转译。与

之前类似，这个 Manifest 描述文件需要以“.nmf”作为后缀，其内部表达的信息需要以 JSON 格式进行描述，具体内容如下。我们在该文件内描述了应用的具体类型（portable），以及需要转译模块的所在位置。

```
pnac1_fibonacci.nmf
{
  "program": {
    "portable": {
      "pnac1-translate": {
        "url": " pnac1_fibonacci.pexe"
      }
    }
  }
}
```

到这里，这个简单的 PNaCl 应用就全部开发完成了。接下来，我们在本地环境中启动一个 HTTP 服务器来对 PNaCl 应用进行测试。本地服务器启动后，我们可以直接在 Web 浏览器中通过对应的本地服务器地址和端口打开之前编写的 index.html 文件。经过 PNaCl 应用从加载运行到计算的一段时间后，我们便可以在浏览器的调试控制台中看到从模块返回的计算结果，如图 1-24 所示。从 C/C++ 层面返回的 pp::VarDictionary 结构会在 JavaScript 环境中自动转换为原生的 JavaScript 对象，这个转换过程也同样是由 Pepper 帮助我们完成的。



图1-24 上述PnaCl应用示例在浏览器中的运行结果

PNaCl 性能基准测试

这里我们将在上述实践过程中得到的测试结果与之前分别对原生 C/C++、ASM.js 和原生 JavaScript 在同样应用下进行的性能测试结果放在一起，并用柱状图进行统计，如图 1-25 所示。可以看到，PNaCl 应用和原生 C/C++ 应用在运行效率上基本保持一致，两者的性能均远超过使用 ASM.js 和原生 JavaScript 实现的应用。

NaCl/PNaCl 技术自 2011 年发展至今已经过去了 7 个年头，虽然像谷歌官方声称的那样，NaCl/PNaCl 应用确实具有很高的性能，但其存在的问题却也十分明显。无论是基于特定处理器架构且不具备可便携性的 NaCl 应用，还是可以被随意在互联网上分发，且同时具有高可用性

的 PNaCl 应用，它们都只能被稳定地运行在 Chrome 浏览器中。在 Google 开发 NaCl 技术的初期，Opera 和 Mozilla 等主流浏览器厂商都认为将更多的精力放在专注于解决 JavaScript 本身和其他现有 Web 技术标准中存在的问题会更有意义，因此他们并没有在各自的浏览器上实现该技术。直到今日，NaCl/PNaCl 技术都没有被除 Chrome 以外的任何其他浏览器支持。

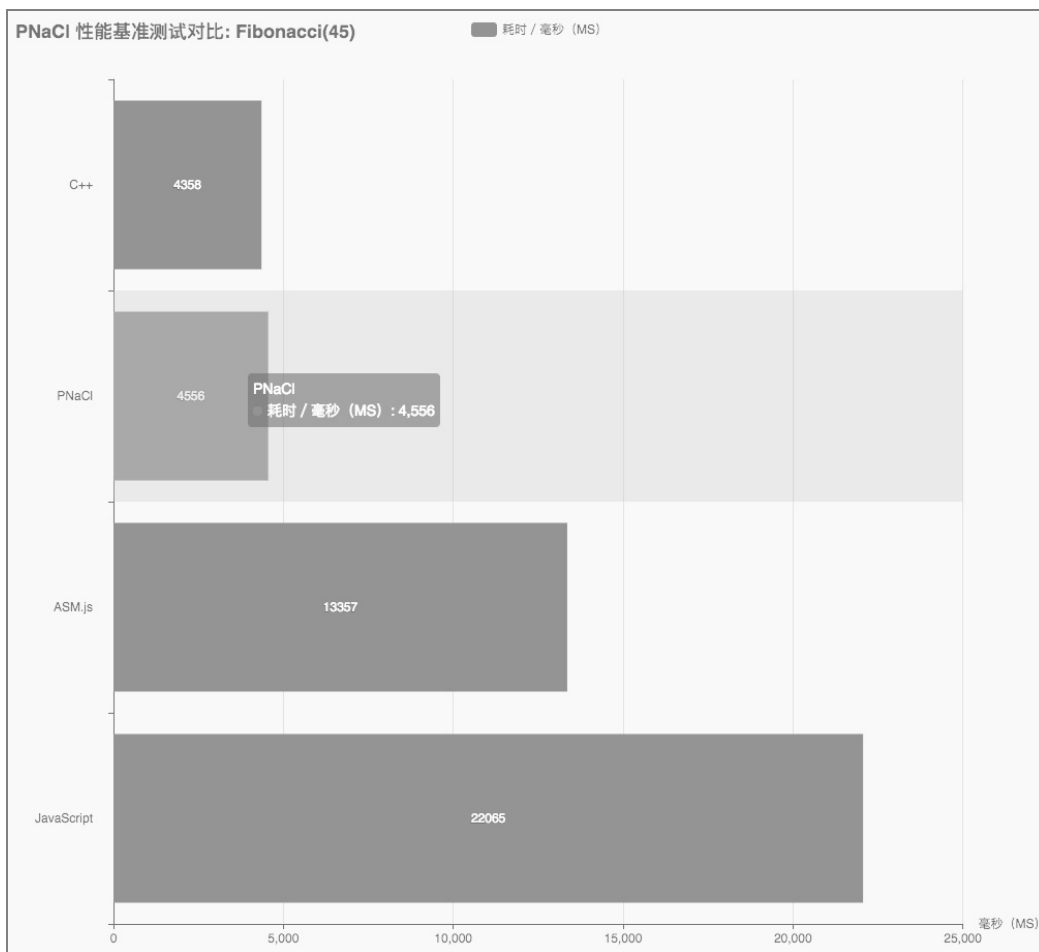


图1-25 PNaCl应用性能基准测试对比

不仅如此，我们也统计了 2017 年 Chrome 浏览器在全球浏览器市场中的用户使用份额，如图 1-26 所示。可以看到，虽然 Chrome 浏览器的市场占有率过半，但仅仅 51.76% 的占有率并不足以让人们可以安心地使用 NaCl/PNaCl 技术来开发 Web 应用，毕竟任何企业和开发者都不会因此而放弃另外 48.24% 的用户。

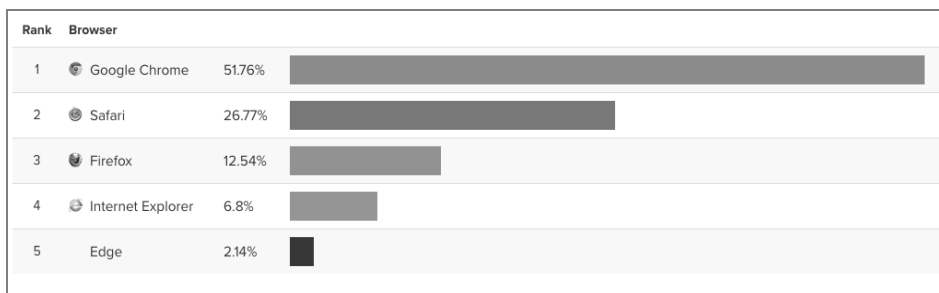


图1-26 2017年各主流浏览器的市场占有率

另一方面，NaCl/PNaCl 应用的使用场景也过于狭窄。高效的音频与视频处理、高性能的计算（如编/解码）需求等应用场景在 Web 领域并不多见。而在互联网上，99%的 Web 网站和应用还是以信息的收集和展示作为其主要功能的。同时，基于 C/C++语言也大大增加了 NaCl/PNaCl 应用的开发难度，这使得 NaCl/PNaCl 基本脱离了技术快速迭代的前端开发领域。而对于中小型企业来说，这种应用的开发成本也过高，浏览器的兼容性问题导致其可能的用户占用率很低，同时用户体验也并不够友好。

由于上面提到的种种原因，加之新技术的出现，谷歌最终于 2016 年 10 月解散了负责维护 NaCl/PNaCl 标准与实现的 Pepper 及 Native Client 团队。2017 年 5 月 30 日，谷歌宣布停止对 PNaCl 技术的维护。从 2018 年的第一季度开始，谷歌将不再支持除 Chrome 官方应用和插件以外的任何 PNaCl 应用。

而谷歌放弃 NaCl/PNaCl 技术的另一个重要原因，便是因为 WebAssembly 出现了。

1.3 新的可能——WebAssembly

ASM.js 与 NaCl/PNaCl 的悲剧收尾并没有让人们放弃去追求构建更高性能的 Web 应用的梦想。通过汲取过去在失败道路上总结的经验与不足，今天 WebAssembly 诞生了。

1.3.1 改变与颠覆

WebAssembly（简称 Wasm）是一种新型的二进制代码格式，包含这种二进制代码格式的文件可以用类似加载模块的方式被浏览器快速、高效地解析和执行。不仅如此，Wasm 并不像 NaCl 应用那样需要区分浏览器运行所在计算机的具体处理器架构，这使得它可以被随意地分发到互联网上，只要位于用户端的浏览器支持 Wasm 格式，用户便可以正常使用其中的功能。

Wasm 的堆栈式机器结构被设计编码成一种高密度、可以被浏览器快速加载和执行的二进制格式。Wasm 的设计目标是，希望浏览器能够以近似原生 C/C++ 应用的运行速度，来调用 Wasm 模块中包含的那些在各类型平台（处理器架构）上都可以使用的通用硬件功能。Wasm 描述了一个内存安全的沙箱执行环境，可以在现有的 JavaScript 虚拟机中进行实现。当在浏览器中运行一个 Wasm 模块时，Wasm 将遵循浏览器中与 Web 应用一致的同源策略来保证其安全性。

为了能够方便地对 Wasm 模块进行调试，Wasm 标准针对 Wasm 的原始二进制格式设计了一种名为“WAT（WebAssembly Text Format）”的可读文本格式，对应于该可读文本格式的文本文件也同样以“.wat”作为后缀。WAT 中的语法结构直接对应着模块的功能和业务逻辑，而这对于深入理解模块的运行机制和对模块的细节进行优化提供了很大的帮助。

Wasm 模块可以通过 JavaScript 提供的上层 Web API 来调用浏览器本身的功能，比如与浏览器在显示的 DOM 内容上进行交互。同时，WebAssembly 不仅可以应用在 Web 浏览器上，它还可以应用在一些非 Web 相关的场景中，比如 Node.js 环境下的后端应用甚至是物联网（IoT）应用。只要这些场景能够提供一个完备的 Wasm 虚拟机便可以正常地解析 Wasm 模块。

基本原理

我们之前介绍过，V8 引擎在解析一段 JavaScript 代码时需要经过一系列的解析器和编译器链路（我们称之为 V8 的 Pipeline 链路）。如图 1-27 所示，JavaScript 会经过链路中各个编译器前置 Parser 的语法分析，最后被转换成 AST 这种中间数据格式，AST 描述了 JavaScript 代码在内存中的语法结构。对应于 Parser 的 Parsing 过程又被分为 Full-Parsing 和 Pre-Parsing 两种类型，这两种过程都需要消耗一定的系统资源。并且，由于 JavaScript 本身是一种弱类型编程语言，在根据 JavaScript 源代码生成对应机器码的过程中，编译器类型推断、优化和去优化等过程也会额外消耗一定的系统资源。

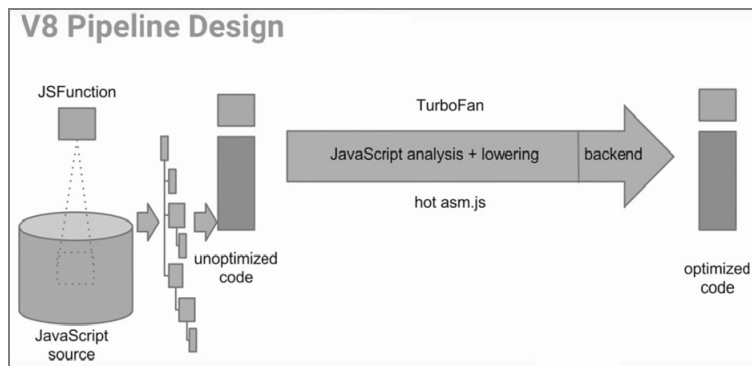


图1-27 Chrome V8编译器链路流程图（图片来源于Google.I/O 2017全球开发者大会）

从图 1-27 中可以看到,从 V8 引擎开始处理 JavaScript 源代码到生成机器码,再到这些机器码最后被浏览器解析和执行,整个过程包含很多道“工序”。在优化编译阶段,TurboFan 优化编译器会使用 IC (Inline Cache) 和 OSR (On Stack Replacement) 等技术来对 JavaScript 源代码进行分析和优化。优化后的代码在生成机器码之前会进行 lowering 操作。在这一步操作中,优化编译器会根据现有的已经优化好的 JavaScript 源代码来生成一些处于低层级且与硬件架构相关的中间代码。而位于整个 V8 链路最末端的编译器后端(backend)就负责将这些经过 lowering 处理的底层中间代码直接转译成基于特定处理器架构的机器码,最后再被浏览器解析和执行。

如果在 Web 浏览器中使用了一个由 Wasm 模块所提供的功能,那么这个 Wasm 模块在整个 V8 链路中从模块加载、解析到最后执行是怎样的流程呢? V8 引擎加载和处理 Wasm 模块的流程如图 1-28 所示。

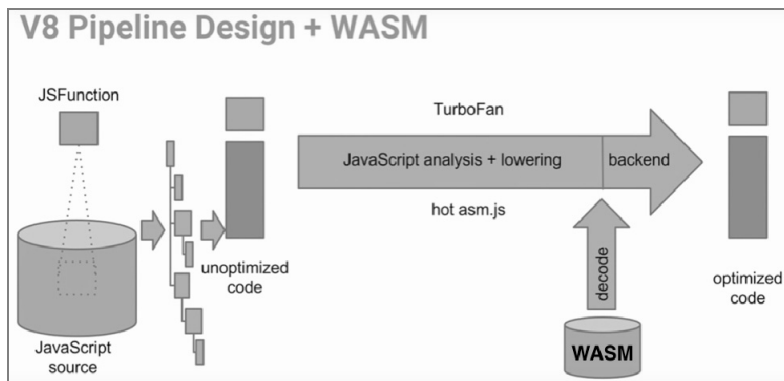


图1-28 Chrome V8+Wasm编译器链路流程图(图片来源于Google.I/O 2017全球开发者大会)

从图 1-28 中可以看到, V8 引擎在处理 Wasm 模块时省略了大量 Pipeline 中的环节。引擎并不需要对 Wasm 模块中的二进制代码进行优化,也不需要生成冗余的占用大量内存的 AST 结构信息。而只需要把这些模块中的二进制代码直接加载到内存中,然后经过位于 V8 链路末端编译器后端的处理,最后生成的机器码便可以浏览器直接执行。从 Wasm 模块被浏览器加载到最后执行的整个过程并不需要很多的处理环节和系统资源开销,而这也是 Wasm 应用为何会保持如此高性能的众多原因之一。

生成与使用

对待任何新技术,我们都本着以“实践第一”为主要原则。那么,在进一步讨论关于 Wasm 更多深入细节信息之前,首先来看一下应该如何生成一个标准的 Wasm 模块,以及在拿到一个 Wasm 模块之后,又该如何在 Web 应用中使用它。

与之前介绍过的 ASM.js 技术类似，在 Wasm 模块描述的程序中，所有变量存储的数据类型都是在程序运行前就已经确定的，并且在程序后续的运行过程中也是无法更改的。V8 链路中的任何一个环节都不会对 Wasm 模块进行优化（代码层面），整个 Wasm 模块的优化工作在编译模块时都已经完成。而浏览器只负责整个 Wasm 生命周期中的最后一个环节，那就是解析并在虚拟机中加载和运行这个 Wasm 模块。

我们可以通过 Emscripten 工具链来生成 ASM.js 代码。同样的，我们也可以使用 Emscripten 工具链来生成一个标准的 Wasm 模块。一般来说，将一个基于 C/C++ 编写的应用程序编译成一个 Wasm 模块有两种常用方法，如图 1-29 所示。

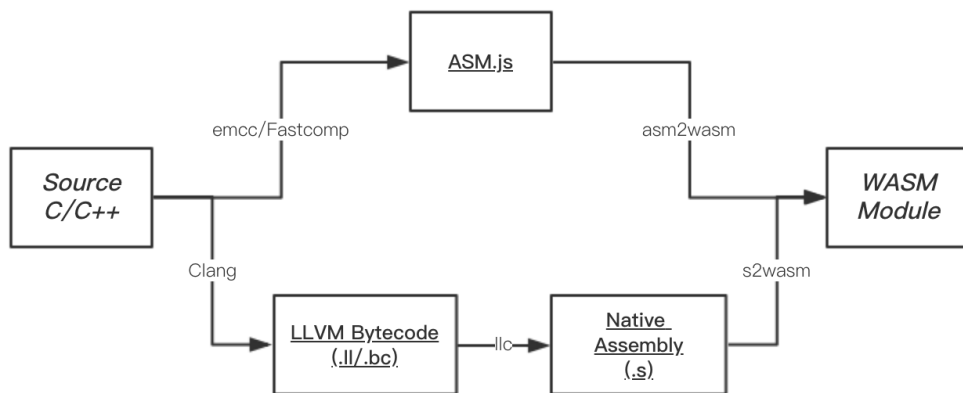


图1-29 构建Wasm模块的两种常用方法

第一种方法，首先将 C/C++ 源代码通过 Emscripten 工具链编译成 ASM.js 代码，然后通过 Binaryen 工具链中的 asm2wasm 工具将这些 ASM.js 代码转译成一个标准的 Wasm 模块。第二种方法，不需要 ASM.js 作为中间媒介，首先直接将 C/C++ 源代码编译成基于 LLVM 的中间比特码，然后再将中间比特码通过 LLVM 工具链编译成与处理器架构相关的本地汇编代码，最后用 Binaryen 工具链中的 s2wasm 工具来作为 Wasm 基于 LLVM 的编译器后端，将基于 LLVM 的本地汇编代码编译成一个标准的 Wasm 模块。

Emscripten 工具链在被设计开发之初，是作为一个专门用于从 C/C++ 源代码编译生成 ASM.js 代码的编译器工具链而存在的。随着 Emscripten 在 ASM.js 上的实践经验越来越多，现阶段通过 Emscripten 工具链将 C/C++ 代码编译为 ASM.js 代码的过程已经十分成熟和稳定。不仅如此，从 C/C++ 等静态语言生成 ASM.js 代码和生成 Wasm 模块的过程也十分相似（两者都是从一种静态强类型语言生成，最后在 Web 浏览器中运行的）。因此，Wasm 官方开发团队在项目初期进行基于 Wasm 最小可用版本（MVP, Minimum Viable Product）标准的编译器开发工作时，

便直接选择了以 Emscripten 作为基础工具链平台，这样从 ASM.js 到 Wasm 的差异化过程便可以通过 asm2wasm 工具中的一个很简单的转译程序来快速实现。

Wasm 在其标准中提到，我们可以使用任意的静态强类型编程语言来编写一个 Wasm 模块，但由于 Wasm 规范刚刚标准化，可用的 LLVM 编译器前端并不多，因此主要还是使用 C/C++ 语言来编写模块所对应的源代码。这里我们将选择上面提到的第一种方式来进行构建第一个 Wasm 应用的实践。

接下来我们会编写一个简单的 Wasm 模块，这个 Wasm 模块的主要功能是可以对一个 JavaScript 数组进行升序排列。我们会在 JavaScript 代码中初始化模块并调用模块向 JavaScript 环境暴露的排序方法，同时会向该方法传递一个事先已经在 JavaScript 环境中生成好的数组对象。在经过 Wasm 模块中排序方法对数组元素进行升序排列的处理后，我们会将数组的排序结果直接打印在浏览器的控制台中。

一个完整的 Wasm 应用的代码组成结构和 PNaCl 应用十分相似。HTML 标签和 CSS 样式负责对网页的展示内容进行布局，而 JavaScript 代码则负责处理页面的交互逻辑，以及加载和连接 Wasm 模块。而 Wasm 模块将会单独从 C/C++ 源代码经过 Emscripten 工具链编译而来。下面先来看一下该 Wasm 模块对应的 C++ 源代码。

```
sort.cc
// 引入 Emscripten 的头文件
#include <emscripten/emscripten.h>
// 引入 C++ STL 库头文件
#include <stack>

using namespace std;

void push2_sequence (stack<double> &stack, double _l, double _r) {
    stack.push(_r);
    stack.push(_l);
}
// 基于栈的快速排序实现
double* quicksort (double sortArray[], double leftPart, double rightPart) {
    stack<double> stack;
    push2_sequence(stack, leftPart, rightPart);
    int _lp, _rp, _mp;

    while(!stack.empty()) {
```

```
double left = stack.top();
stack.pop();
double right = stack.top();
stack.pop();
_lp = left;
_rp = right;
_mp = sortArray[(_lp + _rp) / 2];

while(_lp < _rp) {
    while(sortArray[_lp] < _mp) _lp++;
    while(sortArray[_rp] > _mp) _rp--;

    if(_lp <= _rp) {
        double tmp = sortArray[_lp];
        sortArray[_lp] = sortArray[_rp];
        sortArray[_rp] = tmp;
        _lp++;
        _rp--;
    }
}

if(_lp < right) {
    push2_sequence(stack, _lp, right);
}

if(_rp > left) {
    push2_sequence(stack, left, _rp);
}
}

return sortArray;
}

// 防止出现 C++函数名的 Mangling 问题
#ifdef __cplusplus
extern "C" {
#endif

// 这个向 JavaScript 环境暴露的方法会直接调用模块内部的 quicksort 方法对数组进行排序
// 使用 Emscripten 工具链提供的宏来保证函数不会被编译器进行 DCE 处理
```

```
double* EMSCRIPTEN_KEEPALIVE num_sort (double array[], double length) {
    return quicksort (array, 0, length - 1);
}

#ifdef __cplusplus
}
#endif
```

上面给出的是该 Wasm 模块对应的 C++源代码。与编写普通 C/C++应用程序代码不同的是，在开发一个 Wasm 模块的过程中，会大量使用 Emscripten 工具链提供的一些基于 C/C++语言封装的宏和 API 接口，这些接口可以帮助我们更好地在 C/C++代码中通过虚拟机（VM）环境与 JavaScript 或浏览器进行交互。接下来，我们将深入了解上述源代码中的一些实现细节。

头文件

为了能够在 C/C++源代码中使用 Emscripten 工具链提供的一些特性，我们首先需要在代码中引入 Emscripten 的头文件 `emscripten.h`。这里需要注意设置 Clang/GCC 编译器默认的头文件加载搜索路径，或者直接使用头文件的相对路径来进行加载。

条件编译

接下来便是整个 C++源代码的主体部分。这里主要包括一些与该 Wasm 模块核心业务功能相关的普通 C++函数声明和定义的代码。同时，这里我们使用了条件编译和 `extern` 关键字来防止 C++编译器对函数名的 Mangling 处理行为。

当使用 C++编译器编译一段 C++源代码时，编译器会在当前的编译环境中自动生成一个名为“`__cplusplus`”的默认宏变量。因此，我们可以通过判断该宏变量是否存在，来检查当前正在编译源代码的是否是 C++编译器。而该宏变量所对应的具体值，则会根据 C++编译器所选用的语言编译标准不同而不同。我们可以用如下 C++代码来查看宏变量 `__cplusplus` 所对应的具体值。

```
_cplusplus.cpp
#include <iostream>
using namespace std;

int main() {
    // 打印宏变量__cplusplus 的值
    cout << __cplusplus << endl;
    return 1;
}
```

首先让编译器以默认参数来编译这段 C++ 代码，然后运行编译生成的可执行程序。

```
g++ __cplusplus.cpp -o __cplusplus
./__cplusplus
# 199711
```

接下来让编译器以 C++ 11 标准来编译这段源代码，然后运行编译生成的可执行程序。

```
g++ __cplusplus.cpp -o __cplusplus11 -std=c++11
./__cplusplus11
# 201103
```

可以看到，当编译器使用不同的 C++ 标准来编译目标源代码时，其自动生成的宏变量 `__cplusplus` 对应的值并不相同。而这些值基本都对应着 C++ 各个版本标准的正式发布日期（这里并不一定是标准的准确发布日期，还有可能是通过 C++ 标准委员会最后审批的日期）。

Name Mangling

在用于判断当前编译器类型的条件编译语句中，我们使用 `extern` 关键字指定了一个特殊的作用域。在这个特殊的作用域中，C++ 编译器会强制以 C 语言的语法规则来编译暴露在这个环境内的所有 C/C++ 源代码。

C++ 语言是一种十分灵活的面向对象的编程语言，其独特的函数重载机制使得我们可以轻松地让“类”结构去模拟类似自然界物体的多态行为。函数重载是指在一个作用域内，可以有一组具有相同函数名，但参数列表不同的函数，而这组函数便被称为重载函数。在 C++ 程序中，重载函数通常用来命名一组功能相似的函数，这样做可以减少函数名的数量，避免函数名混用带来的变量污染。同时也增强了程序源代码的可读性。

既然这些重载函数的名字是相同的，那么编译器在编译这些函数时是如何对它们进行区分和处理的呢？我们以如下代码为例，来一探编译器在处理重载函数名时的细节。

```
overload.cc
#include <iostream>

using namespace std;
// 声明一个重载类
class Overload {
private:
    int x;

public:
```

```
// 普通构造函数
Overload (int x) {
    this->x = x;
}
// 拷贝构造函数
Overload (const Overload &o) {
    this->x = o.x;
}

~Overload (){}
// 用于获取该类的私有变量 x 的公有函数
int getX () {
    return this->x;
}
// 类内部的公有函数
int add (int a, int b) {
    return a + b;
}
};

// 普通函数的重载
int add(int a, int b) {
    // 返回两个整数的和
    return a + b;
}

double add(double a, double b) {
    // 返回两个浮点数的和
    return a + b;
}
// 运算符重载
int operator+ (Overload a, Overload b) {
    return a.getX() + b.getX();
}

int main() {
    Overload *a = new Overload(1);
    Overload *b = new Overload(2);
    // 调用重载的运算符
    cout << *a + *b << endl;
    // 调用重载的两个函数
    cout << add(1, 2) << endl;
```

```

cout << add(1.0, 2.0) << endl;
// 调用函数内部的公有方法
cout << a->add(1, 2) << endl;
// 释放内存
delete a;
delete b;
return 0;
}

```

接下来，我们使用 GCC 编译器来编译这段 C++ 代码并生成以 “.o” 为后缀的中间目标对象文件，然后通过 Linux 系统下的 `gobjdump` 或 `nm` 命令来查看该对象文件内的符号表信息。符号表是一种常用在编译器或解释器中的数据结构。在符号表中，程序源代码中的每个标识符都和它的声明或使用信息绑定在一起，比如源代码中变量的数据类型、作用域及内存地址。符号表在编译器的工作过程中不断收集和记录程序源代码在语法和语义上的特征和信息。因此在符号表中，我们可以直接看到编译器在处理重载函数后得到的中间结果。

```

// 编译源代码并生成中间目标对象文件
g++ overload.cc -o overload.o
// 查看对象文件中的符号表信息
gobjdump -x overload.o

```

命令运行后得到的结果如图 1-30 所示。

```

SYMBOL TABLE:
00000000100000bc0 g 1e SECT 01 0080 [.text] __ZN8OverloadC1Ei
00000000100000bf0 g 1e SECT 01 0080 [.text] __ZN8OverloadC1ERKS_
00000000100000c20 g 1e SECT 01 0080 [.text] __ZNSt3_14endlIcNS_11char_traitsIcEEEEERN5_13basic
00000000100000d30 g 1e SECT 01 0080 [.text] __ZN8OverloadD1Ev
00000000100000d50 g 1e SECT 01 0080 [.text] __clang_call_terminate
00000000100000d80 g 1e SECT 01 0080 [.text] __ZN8OverloadC2Ei
00000000100000da0 g 1e SECT 01 0080 [.text] __ZN8OverloadC2ERKS_
00000000100000dc0 g 1e SECT 01 0080 [.text] __ZN8OverloadD2Ev
00000000100000e84 l 0e SECT 04 0000 [__TEXT,__gcc_except_tab] GCC_except_table5
00000000100000f3c l 0e SECT 04 0000 [__TEXT,__gcc_except_tab] GCC_except_table8
000000001000008d0 g 0f SECT 01 0000 [.text] __Z3adddd
000000001000008b0 g 0f SECT 01 0000 [.text] __Z3addii
000000001000008f0 g 0f SECT 01 0000 [.text] __Z7charSeqPc
00000000100000d60 g 0f SECT 01 0080 [.text] __ZN8Overload3addEii
00000000100000930 g 0f SECT 01 0080 [.text] __ZN8Overload4getXEv
00000000100000900 g 0f SECT 01 0000 [.text] __Zpl8OverloadS
00000000100000000 g 0f SECT 01 0010 [.text] __mh_execute_header
00000000100000940 g 0f SECT 01 0000 [.text] _main
00000000000000000 g 01 UND 00 0200 __Unwind_Resume
00000000000000000 g 01 UND 00 0100 __ZNKSt3_16locale9use_facetERN5_12idE
00000000000000000 g 01 UND 00 0100 __ZNKSt3_18ios_base6getLocEv
00000000000000000 g 01 UND 00 0100 __ZNSt3_113basic_ostreamIcNS_11char_traitsIcEEEE3putEc
00000000000000000 g 01 UND 00 0100 __ZNSt3_113basic_ostreamIcNS_11char_traitsIcEEEE5flushEv
00000000000000000 g 01 UND 00 0100 __ZNSt3_113basic_ostreamIcNS_11char_traitsIcEEEE5sEi
00000000000000000 g 01 UND 00 0100 __ZNSt3_14coutE
00000000000000000 g 01 UND 00 0100 __ZNSt3_15ctypeIcE2idE
00000000000000000 g 01 UND 00 0100 __ZNSt3_16localeD1Ev
00000000000000000 g 01 UND 00 0100 __ZSt9terminatev
00000000000000000 g 01 UND 00 0180 __ZdlPv
00000000000000000 g 01 UND 00 0180 __Znmw
00000000000000000 g 01 UND 00 0100 __cxa_begin_catch
00000000000000000 g 01 UND 00 0100 __gxx_personality_v0
00000000000000000 g 01 UND 00 0200 dyld_stub_binder

```

图1-30 通过 `gobjdump` 命令打印的符号表信息

从符号表中可以看到，编译器在处理 C++源代码时直接把其中重载的两个名为“add”的函数的名称分别替换成了“__Z3adddd”与“__Z3addii”。不仅如此，在 Overload 这个 C++类中声明的公有函数 getX 的名称也被编译器处理并替换成了“__ZN8Overload4getXEv”。这种在编译阶段重载函数的函数名被重新拆分和重组的现象，便是由 C++开发中常见的 Name Mangling 机制导致的。但也正是因为该机制的加入，才使得 C++语言能够支持函数重载的特性。

当 C++编译器在处理 C++源代码时，会分析每一个声明函数的返回值类型、函数名和参数列表。综合这三方面信息，编译器会为每一个函数重新生成一个新的函数名。这样就能保证每一个函数都能拥有自己唯一的函数签名，编译器也可以通过这个唯一的函数签名来调用该函数。另一方面，在编译器最上层的 C++源代码中，重载的特性也可以被很好地支持。

经过 Name Mangling 机制生成的函数名一般由几个部分组成，这里以上面 C++源代码中的 add 函数为例。比如其中一个 add 函数的参数列表是由两个整数类型的参数组成的，同时该函数最后的返回值也是一个整数类型，那么编译器对该函数进行 Name Mangling 处理的流程是：首先，编译器会对该函数的参数列表进行缩写处理，即缩写成“ii”这种形式（“i”是声明整型变量关键字 int 的首字母）。在进行缩写处理之后，编译器会把这个参数列表的缩写形式拼接到函数名的后面，此刻函数名便变成了“addii”。接下来，编译器会开始处理函数的返回值类型。对于每一种具体的返回值类型，编译器都会使用一个特定的数字来表示。比如在 GCC 编译器中，数字“3”表示该函数的返回值是数值类型（整数或浮点数）。在处理完函数的返回值类型后，编译器会将这些表示函数各个部分特征的缩写拼接在一起，同时再加上一些编译器本身自定义的元信息，最后便形成了经过 Name Mangling 处理后的符号函数名“__Z3addii”。

对于一些在更复杂场景下函数的 Name Mangling 操作，编译器也会有着更复杂的函数名转换规则与之相对应。比如在上面的 C++源代码中，对于声明在“类”结构中的一个同样名为“add”且函数体内容也相同的类成员函数，编译器在对它进行 Name Mangling 处理后，其符号表中的符号函数名变成了“__ZN8Overload3addEii”。总而言之，编译器会通过 Name Mangling 机制让源代码中的每一个函数声明都有一个与之相对应的唯一函数名，这样便可以保证编译器在链接目标文件时，能够根据这个唯一的函数签名准确地找到对应的函数体。

```
extern "C">{ ... }
```

为了能够支持 C++语言多态特性中的重载机制，C++编译器在编译 C++源代码声明的函数时，会通过 Name Mangling 机制为每一个函数根据其特征重新生成一个唯一的符号函数名。这个重新生成的函数名会被临时存储在编译器的符号表中，而编译器则会根据符号表中的信息来

生成 C++ 源代码所对应的抽象语法树（AST）结构。

那么如何才能不让 C++ 编译器对 C++ 源代码中的函数名进行 Name Mangling 处理，从而让位于抽象语法树结构中的函数直接使用其对应源代码中的原始函数名？此时我们便可以通过 `extern` 关键字来解决这个问题。

“`extern "C" { ... }`” 语句块可以让编译器以 C 语言的规则来处理位于其作用域内的代码。由于 C 是一种面向过程的编程语言，其语法规则中并不存在多态这种特性。因此，当 C++ 编译器以 C 语言的规则来处理位于该语句块作用域内的代码时，并不会对其中声明的函数进行 Name Mangling 处理，而是直接使用开发者在 C++ 源代码中定义的函数名作为抽象语法树结构中的函数名。我们可以通过下面给出的代码，来看一下 “`extern "C" { ... }`” 语句块在编译代码时对处于其作用域内部的函数会产生怎样的作用。

```
extern_c_mangling.cc
#include <iostream>

using namespace std;
// 位于 extern 语句外部的代码，编译器会以 C++ 的语法规则来编译
void manglingFunction() {
    ;
}
// 位于 “extern "C" { ... }” 语句块内的代码，编译器会按照 C 语言的规则来编译
extern "C" {
    int noManglingFunction (int a, int b) {
        return a + b;
    }
}

int main() {
    // 调用函数，防止函数被 DCE 过程处理
    manglingFunction();
    cout << noManglingFunction(1, 1) << endl;
}
```

我们将上述源代码编译成一个以“.o”为后缀的中间目标对象文件。然后同样使用 `gobjdump` 命令来查看编译器在编译这段 C++ 源代码时生成的符号表信息，如图 1-31 所示。


```

SYMBOL TABLE:
0000000100000d90 g    1e SECT  01 0080 [.text] __ZNSt3_14endlcNS_11char_traitsIcEEEEENS_13bas
0000000100000ea0 g    1e SECT  01 0080 [.text] ___clang_call_terminate
0000000100000f54 l    0e SECT  04 0000 [__TEXT,__gcc_except_tab] GCC_except_table3
0000000100000d10 g    0f SECT  01 0000 [.text] __Z16manglingFunctionv
0000000100000000 g    0f SECT  01 0010 [.text] __mh_execute_header
0000000100000d40 g    0f SECT  01 0000 [.text] _main
0000000100000d20 g    0f SECT  01 0000 [.text] _noManglingFunction
0000000000000000 g    01 UND   00 0200 __Unwind_Resume
0000000000000000 g    01 UND   00 0100 __ZNKSt3_16locale9use_facetERS0_2ide
0000000000000000 g    01 UND   00 0100 __ZNKSt3_18ios_base6getlocEv
0000000000000000 g    01 UND   00 0100 __ZNSt3_113basic_ostreamIcNS_11char_traitsIcEEEE3putEc
0000000000000000 g    01 UND   00 0100 __ZNSt3_113basic_ostreamIcNS_11char_traitsIcEEEE5flushEv
0000000000000000 g    01 UND   00 0100 __ZNSt3_113basic_ostreamIcNS_11char_traitsIcEEEE5ei

```

图1-31 通过gobjdump命令打印的符号表信息

从符号表中列出的符号信息可以看到,位于“extern "C" { ... }”语句内的 noManglingFunction 函数并没有被 C++编译器进行 Name Mangling 处理,其位于符号表中的函数名并没有发生任何变化。而对于位于该语句外面的 manglingFunction 函数,其位于符号表中的函数名则被编译器替换成了“__Z16manglingFunctionv”。这便是支持多态的强类型语言所独有的 Name Mangling 特性。

EMSCRIPTEN_KEEPLIVE

EMSCRIPTEN_KEEPLIVE 是 Emscripten 工具链提供的一个宏,通过使用该宏我们可以防止 C++编译器在处理和优化 C++代码时,将那些没有使用到的函数或代码段消除,即进行 DCE (Dead Code Elimination)处理。我们还是通过下面给出的 C++代码来说明 DCE 的具体工作流程。

```
dce.cc
```

```

#include <iostream>

using namespace std;
// 定义一个全局函数
int add (int a, int b) {
    return a + b;
}

int main() {
    // 声明一个名为 s 的长整型变量
    long long s = 0;
    // 一个超长的循环语句,让变量 s 的值和 i 不断地进行累加
    for (long long i = 1; i <= 10000000000; ++i) s += i;
    // 打印语句
    cout << "Print some codes." << endl;
}

```

接下来，我们使用 GCC 编译器来编译这段 C++ 代码。在编译过程中，我们为编译器添加了一个名为 “dump-tree-optimized-graph” 的标志，该标志能够让编译器输出优化阶段的中间表示代码。

```
// 编译源代码并输出优化阶段的中间表示代码
g++ -fdump-tree-optimized-graph dec.cc -o dec
```

当 GCC 编译器对代码编译完成后，会在当前目录下生成一个名为 “dce.cc.165t.optimized” 的文件。在该文件内，编译器以中间代码的形式表示了 C++ 源代码经过 GCC 优化后的某个中间代码。该文件的部分内容如下。

```
;; Function int add(int, int) (_Z3addii, funcdef_no=969, decl_uid=20767, cgraph_uid=213)
;; 对应 C++ 源代码中的全局 add 函数
int add(int, int) (int a, int b)
{
    int D.20796;
    int _3;

    <bb 2>:
        _3 = a_1(D) + b_2(D);

<L0>:
    return _3;
}

;; Function int main() (main, funcdef_no=970, decl_uid=20769, cgraph_uid=214)

int main() ()
{
    struct basic_ostream & D.20801;
    ;; 对应 C++ 源代码主函数中的循环语句
    long long int i;
    long long int s;
    int D.20799;
    struct basic_ostream & D.20798;
    struct basic_ostream & _7;
    struct basic_ostream & _8;
    int _9;

    <bb 2>:
```

```

s_3 = 0;
i_4 = 1;
goto <bb 4>;

<bb 3>:
s_5 = s_1 + i_2;
i_6 = i_2 + 1;

<bb 4>:
# s_1 = PHI <s_3(2), s_5(3)>
# i_2 = PHI <i_4(2), i_6(3)>
if (i_2 <= 1000000000)
    goto <bb 3>;
else
    goto <bb 5>;

;; 对应 C++ 源代码中的输出打印语句
<bb 5>:
_7 = std::operator<< <std::char_traits<char> > (&cout, "Print some codes.");
_8 = _7;
std::basic_ostream<char>::operator<< (_8, endl);
_9 = 0;

<L3>:
    return _9;
}

;; Function void __static_initialization_and_destruction_0(int, int) (_Z41__static_
initialization_and_destruction_0ii, funcdef_no=979, decl_uid=20788, cgraph_uid=223)

void __static_initialization_and_destruction_0(int, int) (int __initialize_p, int
__priority)
{
    <bb 2>:
    if (__initialize_p_1(D) == 1)
        goto <bb 3>;
    else
        goto <bb 5>;

    <bb 3>:

```

```
if (__priority_2(D) == 65535)
    goto <bb 4>;
else
    goto <bb 5>;

<bb 4>:
std::ios_base::Init::Init (&__ioinit);
__cxa_atexit (__comp_dtor , &__ioinit, &__dso_handle);

<bb 5>:
return;
}

;; Function (static initializers for dce.cc) (_GLOBAL__sub_I__Z3addii, funcdef_no=980,
decl_uid=20794, cgraph_uid=224)

(static initializers for dce.cc) ()
{
    <bb 2>:
    __static_initialization_and_destruction_0 (1, 65535);
    return;
}
```

可以看到，如果使用 GCC 编译器默认的优化策略来编译和优化代码，并不会对原始的 C++ 代码逻辑有任何影响。接下来我们重新编译这段代码，在编译时我们会为 GCC 指定一个编译优化参数“-O3”，让编译器以一个比较深度的优化策略来编译和优化这段代码，命令如下。

```
g++ -O3 -fdump-tree-optimized-graph dec.cc -o dec
```

代码编译完成后，我们再来看一下 dce.cc.165t.optimized 文件的内容。

```
;; Function int add(int, int) (_Z3addii, funcdef_no=973, decl_uid=20786, cgraph_uid=217)
// 在全局作用域中声明的函数
int add(int, int) (int a, int b)
{
    int _3;

    <bb 2>:
    _3 = a_1(D) + b_2(D);
    return _3;
}
```

```

;; Function int main() (main, funcdef_no=974, decl_uid=20788, cgraph_uid=218) (executed
once)

int main() ()
{
    struct basic_ostream & _3;
    // 对程序运行结果无任何影响的循环逻辑块被 DCE 处理掉
    <bb 2>:
    // 主函数中的打印语句
    _3 = std::operator<< <std::char_traits<char> > (&cout, "Print some codes.");
    std::endl<char, std::char_traits<char> > (_3);
    return 0;

}

;; Function (static initializers for dce.cc) (_GLOBAL__sub_I__Z3addii, funcdef_no=984,
decl_uid=20813, cgraph_uid=228) (executed once)

(static initializers for dce.cc) ()
{
    <bb 2>:
    std::ios_base::Init::Init (&__ioinit);
    __cxa_atexit (__comp_dtor , &__ioinit, &__dso_handle); [tail call]
    return;
}

```

可以看到，经过 GCC 以深度优化策略（-O3）优化后的中间代码比使用默认优化策略产生的中间代码少了很多。如果仔细查看这些中间代码，就会发现少了一部分主函数中的逻辑代码。原先位于 C++ 主函数中的“超长循环累加语句”代码被去掉了，而这便是 C++ 代码经过编译器 DCE 处理后的结果。编译器在进行比较深度的代码优化时，会分析源代码中那些对程序的最终运行结果没有影响的逻辑过程，比如位于 C++ 主函数中的循环累加逻辑。经过循环累加所得到的变量“s”的值并没有被应用到主程序中的任何地方，编译器在进行代码优化时会认为这段逻辑对程序的最终运行结果是没有任何影响的，因此借由 DCE 策略对代码进行优化后将这部分代码移除。这样经过处理后，可以让程序更加高效地运行，同时编译后生成的可执行文件体积也会相应地减小。

进一步的，我们可以通过为 gcc 命令增加一些额外的参数，来让 GCC 编译器以更加深度的 DCE 方式来优化源代码。在命令行中运行如下命令。

```
gcc -Os -fdata-sections -lstdc++ -ffunction-sections dce.cc -o dce.o -Wl,--gc-sections -Wl,--print-gc-sections
```

在这行命令中，我们为 GCC 编译器指定了“-fdata-sections”和“-ffunction-sections”两个参数。通过这两个参数，编译器可以将源代码中出现的所有函数和变量设置成独立的 ELF 分段。独立的 ELF 分段意味着当分段的内容没有被链接器或汇编器使用时，便可以将其从源程序中移除。命令中的“-gc-sections”便是告知编译器在编译和链接时将没有使用到的 ELF 分段进行垃圾回收。通过使用“-print-gc-sections”参数，编译器可以在编译过程中将被垃圾回收掉的 ELF 分段信息打印出来，如图 1-32 所示。

```
[root@iZ23ki8vt8tZ ~]# gcc -Os -fdata-sections -lstdc++ -ffunction-sections dce.cc -o dce.o -Wl,
/usr/bin/ld: Removing unused section '.note.ABI-tag' in file '/usr/lib/gcc/x86_64-redhat-linux/4
/usr/bin/ld: Removing unused section '.rodata.cst4' in file '/usr/lib/gcc/x86_64-redhat-linux/4.
/usr/bin/ld: Removing unused section '.data' in file '/usr/lib/gcc/x86_64-redhat-linux/4.8.2../
/usr/bin/ld: Removing unused section '.text._Z3addii' in file '/tmp/cc537oNo.o'
```

图1-32 GCC在进行ELF分段垃圾回收时的相关信息

图中名为“.text._Z3addii”的 ELF 分段便对应着我们在 C++源代码中定义的那个名为“add”的全局函数。在这里由于该函数没有被主函数中的任何逻辑代码引用，因此，编译器在进行 ELF 分段垃圾回收时，会将该函数对应的 ELF 分段直接丢弃。以上便是 GCC 编译器在对 C++源代码进行 DCE 优化时各个阶段的详细过程。

回过头来看，当使用 Emscripten 工具链从 C++源代码编译生成一个 Wasm 模块时，Emscripten 也会通过 DCE 这种方式来减小模块的体积。Emscripten 会首先将 C++源代码编译成 LLVM 中间代码的形式。在进行代码优化时，Emscripten 会调用 LLVM 工具链中的优化器对这些中间代码进行深度优化。同样的，那些对主程序最终运行结果没有任何影响的逻辑代码，以及在模块主流程中没有被调用到的函数定义，也都会在 LLVM 优化器的处理过程中被去掉。

在通常情况下，我们会将单独的 Wasm 二进制文件作为一个独立的模块来进行使用（即模块对外暴露出一些特定的接口，并且可以方便地在 JavaScript 环境中调用），而此时该 Wasm 模块对应的 C++源代码中通常是不包含 main 入口函数的。因此，我们需要通过 Emscripten 提供的宏参数 EMSCRIPTEN_KEEPALIVE 来对需要导出到 JavaScript 环境中的方法做出标记，以防止其在编译器的 DCE 过程中被优化器处理掉。关于带有 C/C++主函数的 Wasm 模块，我们会在专门讲解 Emscripten 工具链的章节中再进行深入介绍。

接下来，我们可以通过 Emscripten 工具链提供的 CLI 命令行工具将这段 C++代码编译成一个标准的 Wasm 模块，命令如下。

```
sudo emcc -s WASM=1 -O3 -o sort.js sort.cc
```

上述命令直接调用了 Emscripten 工具链中“Emscripten 编译器前端 (Emscripten Compiler Frontend)”所对应的命令行工具 `emcc`。在这条命令行语句中，我们通过“-s”参数为 `emcc` 命令指定了一个特殊的标志位“WASM=1”。通过设置该标志位，`emcc` 会在编译 C++ 源代码并生成其所对应的 `ASM.js` 代码后，再通过调用 Binaryen 工具链提供的 `asm2wasm` 工具来将 `ASM.js` 代码转译成标准的 `Wasm` 模块。即对应着我们之前总结的如何将 C++ 代码编译到 `Wasm` 模块中的第一种方法。

命令执行完毕后，`emcc` 会在当前文件目录下生成两个文件。其中一个是以“.wasm”为后缀的目标模块文件；另一个则是 Emscripten 自动生成的用于进行“JavaScript Binding”的 `JavaScript` 脚本文件。在这个 `JavaScript` 脚本文件中，Emscripten 已经自动生成了用于在 `JavaScript` 环境下加载及初始化该 `Wasm` 模块所需要的所有环境变量、参数和方法。除此之外，一些用于控制和操作 `Wasm` 模块底层特性（内存与文件系统等）的功能函数也被内置其中。接下来，我们需要编写一个 `HTML` 页面，然后通过这个 `JavaScript` 脚本文件来将 `Wasm` 模块间接地加载到浏览器中，最后再调用模块向 `JavaScript` 环境中暴露出的方法，并将其执行结果打印在浏览器的控制台中。`HTML` 文件的具体内容如下。

```
<DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>第一个 WebAssembly 应用</title>
  </head>
  <body>

    <script>
      // 初始化一个全局的 Module 对象，Emscripten 自动生成的 JavaScript Binding 脚本
      // 会自动填充该对象中的内容
      var Module = {};
      // 声明一个脚本标签，用于动态加载 JavaScript Binding 脚本文件
      var script = document.createElement('script');

      // 从远程加载 Wasm 模块，然后获取模块内部的字节数据
      fetch('sort.wasm').then(response => response.arrayBuffer()).then((bytes) => {
        // 填充模块的主要数据到 Module 对象的初始化属性
        // JavaScript Binding 脚本会通过 Module 对象中的 wasmBinary 属性来初始化一个 Wasm 模块实例
        Module.wasmBinary = bytes;
        // 异步载入用于初始化的 JavaScript Binding 脚本
```

```

    script.src = "sort.js";
    document.body.appendChild(script);
  });

// 当 JavaScript Binding 脚本被完全加载后执行的回调函数
script.onload = function () {
  // 脚本加载成功后, 此时的 Module 对象已经完成了初始化, Wasm 模块向 JavaScript 环境暴露的
  // 方法已经被全部加载到 Module 对象中, 接下来便可以调用模块暴露出的方法
  let call_array_memeory_method = (method, array) => {
    // 声明一个 double 类型在内存中占用的比特数
    const _sizeof_double = 8
    // 计算待排序数组的长度 (以比特为单位)
    var _size = array.length * _sizeof_double;
    // 手动分配一块内存, 返回值 "_buff" 为新分配内存的首地址
    var _buff = Module._malloc(_size);
    // 将数组中的数据拷贝到这块新分配的内存中
    for (var i = 0; i < array.length; i++) {
      // 注: 在新版本 Emscripten 工具链中, 若要使用该方法, 可能需要在编译时通过为 emcc 指定
      // "-s 'EXTRA_EXPORTED_RUNTIME_METHODS=[\"setValue\"]'" 参数来将其显式地导出
      Module.setValue(_buff + _sizeof_double * i, array[i], 'double');
    }
    // 调用 Wasm 模块暴露出的方法对数组进行排序, 同时返回排序后生成的新数组在内存中的首地址
    var _offset_buff = method(_buff, array.length);
    var result = [];
    // 重新从该地址中依次读出新数组的数据项
    for (var i = 0; i < array.length; i++) {
      // 注: 在新版本 Emscripten 工具链中, 该方法同样需要被显式导出, 导出方式同 setValue
      result.push(Module.getValue(_offset_buff + _sizeof_double * i, 'double'));
    }

    // 释放之前手动创建的内存段
    Module._free(_buff);
    // 返回最后的排序结果
    return result;
  }

  // 主排序函数
  let sort = array => {
    // 从 Module 对象中调用 Wasm 模块对外暴露出的 JavaScript 接口方法 _num_sort

```



```
        return call_array_memeory_method(Module["asm"]["_num_sort"], array);
    }

    let array = [5, 7, 1, 4, 6, 7, 9, 0];
    console.log(sort(array));
}
</script>
</body>
</html>
```

创建好对应的 HTML 文件后，还需要在当前目录下启动一个小型的 HTTP 服务器来提供 HTTP 服务。服务器启动后，便可以通过 Chrome 浏览器载入该 HTML 页面。经过一段时间的 Wasm 模块加载和函数调用过程后，我们便可以在该页面对应的浏览器控制台中看到最后的数组排序结果。接下来，我们将对上述代码的部分实现细节进行详细说明。

异步加载脚本

我们需要将 Emscripten 工具链生成的 JavaScript Binding 脚本异步加载到当前的 HTML 文件中。当脚本中的 JavaScript 代码被浏览器解析执行时，脚本会首先检查在当前的 JavaScript 作用域内是否存在名为“Module”的原生 JavaScript 对象变量，若存在，则会直接使用该对象中名为“wasmBinary”的属性来初始化 Wasm 模块；否则，脚本将会尝试从当前位置直接加载带有默认文件名的 Wasm 模块。而在 wasmBinary 属性中，则以二进制比特数据的形式存储着用于初始化一个 Wasm 模块的完整数据内容。

当 JavaScript Binding 脚本中的代码完成对 Wasm 模块的初始化操作时，脚本会将模块暴露给 JavaScript 环境的所有可用方法整合到 Module 对象中。因此，只有当 Module 对象完成初始化后，我们才能进行后续模块方法调用过程。为此，需要把这些在 Wasm 模块完成初始化后才能够进行调用的代码，全部放到 script.onload 这个脚本加载完成的回调方法里（不支持 IE），并以 script 标签异步加载 JavaScript 脚本的方式来动态加载 Emscripten 生成的 JavaScript Binding 脚本文件。script.onload 方法会等待脚本内所有的 JavaScript 代码全部执行完毕后，才会执行回调方法内的代码，而此时 Wasm 模块的初始化已经完成。

Module 对象

在加载 JavaScript Binding 脚本文件之前，我们首先在全局 JavaScript 作用域内声明了一个名为“Module”的原生 JavaScript 对象，这个对象结构在接下来的过程中会被 JavaScript Binding 脚本作为其初始化时的方法容器。脚本初始化后，会将其内部用于操作内存、浏览器交互，以及 Wasm 模块本身对外暴露出的所有方法全部挂载到这个原生的 JavaScript 对象结构中。

接下来, 我们使用 `fetch` 方法从远程加载一个标准的 Wasm 模块文件。当模块加载完成后, 再通过 `FETCH API` 标准定义的用于处理其响应数据的方法将这次请求的响应数据转换成 `ArrayBuffer` 的二进制数据形式。在这个 `ArrayBuffer` 结构中, 以二进制比特数据的形式存储着上一步从远程加载好的整个 Wasm 模块内容。这里也可以使用传统的 `AJAX` 方法来加载模块, 只要确保最后可以将这些从远程获取到的 Wasm 模块内容转换成 `ArrayBuffer` 的二进制形式即可。

随后, 我们使用这些已经转换好的二进制比特数据来“填充”`Module` 对象中名为 `wasmBinary` 的属性。当 `wasmBinary` 属性被填充完毕后, 便可以开始动态加载这个用于在 `JavaScript` 环境下初始化 Wasm 模块的 `JavaScript Binding` 脚本文件了。这里直接将之前创建的 `script` 标签的 `src` 属性指定为脚本文件的所在位置, 然后将该标签整体拼接到当前的 `HTML` 文档中即可。

当新的 `script` 标签被拼接到 `HTML` 文档中时, 浏览器便开始从远程位置加载 `script` 标签指定的 `JavaScript` 脚本文件。加载完成后, 浏览器开始执行脚本内部的代码。当脚本内的代码被执行完毕后, 该 `script` 标签对应的 `script.onload` 方法便会被调用执行。而在这个回调函数的内部, 我们便可以通过 `Module` 对象与已经加载并初始化好的 Wasm 模块进行交互。

通过共享内存传递数组

在 `script.onload` 对应的回调函数中, 我们主要调用 Wasm 模块暴露出的 `num_sort` 方法来对 `JavaScript` 中声明的一个数组进行了排序。在排序完成后, 将排序结果打印到浏览器控制台。

对于数组和对象这类相对比较复杂的 `JavaScript` 数据类型, 我们则需要通过使用 `TypedArray` 这种数据结构来在 `JavaScript` 环境和 Wasm 模块之间传递数据。`TypedArray` 可以为我们在两种不同的运行时环境之间开辟出一个共享的内存空间。在这个内存空间中, 我们可以从 `JavaScript` 环境中将数据填充进去, 然后在 Wasm 模块的运行时环境中再通过这个共享内存空间的首地址 (指针) 和存储数据的具体类型将数据依次取出。基本流程如图 1-33 所示。

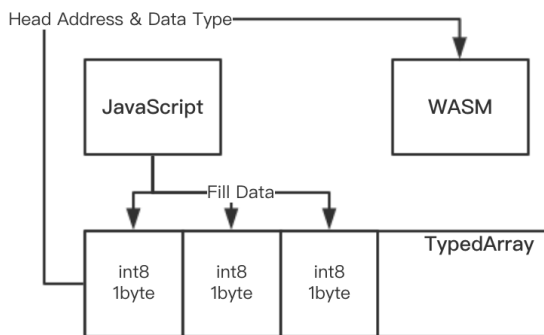


图1-33 在JavaScript环境与Wasm模块之间传递复杂类型数据的基本流程

当使用 `TypedArray` 这种数据结构来进行数据存储时，需要区分所存储数据的具体类型，这使得 `JavaScript` 可以更容易地与 `Wasm` 模块包含有强类型数据的内存环境进行交互。在编写该 `Wasm` 模块时的 C++ 源代码中，我们已将排序方法接受的数组成员设置为双精度浮点型，因此，当从 `JavaScript` 环境向 `TypedArray` 中填充数据时也需要选择对应于双精度浮点型的 `TypedArray` 类型，比如这里应该选择 `Float64Array` 类型的 `TypedArray`。但好消息是，`Emscripten` 工具链已经为我们准备好了这些常用的数据操作方法。在这个由 `JavaScript Binding` 脚本填充好的 `Module` 对象中，已经提供了用于生成和编辑基于共享内存段 `TypedArray` 数据结构的方法 `setValue`，我们可以直接使用而不必再单独进行封装。

从代码中可以看到，我们通过 `Module` 对象内置的 `_malloc` 方法开辟出了一个长度为“数组元素个数×单个数组元素占用内存字节数”的共享内存空间。在这里数组长度为 8，数组中的每个元素在 `Wasm` 模块里都会被转换成双精度浮点型，因此每个元素占用 8 字节大小的内存，所以整个数组元素占用内存大小即为 64 字节。

`Module._malloc` 方法在分配完内存空间后会立即返回该段共享内存的首地址。接下来，我们便可以通过 `Module` 对象中的 `setValue` 方法来向该段内存中写入数据。`setValue` 方法接收两个参数：第一个参数为需要存储的数据在内存中的位置，即偏移的内存地址；第二个参数为需要存储的数据值。对于第一个参数，需要在每次存储数据时都在之前 `Module._malloc` 方法返回的首地址上进行逐次累加，即每次填充一个数据后将该地址向内存高位增加 8 字节，也就是一个双精度浮点型变量在内存中的大小。

当通过循环语句将数据全部写入共享内存后，便可以通过 `Module` 对象来调用 `Wasm` 模块向 `JavaScript` 环境暴露的方法了。`Wasm` 模块暴露出的所有方法全部被挂载在 `Module` 对象内部的 `asm` 对象中。这里我们直接调用了该对象上的 `_num_sort` 方法，同时将包含有数组元素数据的共享内存首地址和数组元素个数作为参数传入。

当 `Wasm` 模块将共享内存中的数据处理完毕后，我们还需要使用相同的方式从 `JavaScript` 环境将共享内存中的数据结果提取出来。这里直接使用 `Module` 对象提供的 `getValue` 方法，在调用该方法时同样需要传入数据所在的内存偏移地址，以及数据对应的类型字符串。`getValue` 方法会自动根据类型字符串对应的数据类型在共享内存中读取数据。当所有的数据处理与交换过程结束后，我们还需要手动将之前从 `JavaScript` 环境创建的共享内存段释放掉，这里直接使用 `Module` 对象内置的 `_free` 方法来释放内存，以防止内存泄漏。

至此，一个基于 `Wasm` 的复杂数据类型处理应用便完成了。对于一些简单的 `JavaScript` 数据类型，可以省略掉创建共享内存段的过程。另外，对于如何基于 `Emscripten` 工具链创建更加

负责的前端 Web 应用，我们会在后续的专题章节中进行介绍。

性能基准测试

这里还是使用之前的那个用于推导斐波那契数列的应用来作为本次 Wasm 性能基准测试的基本方案。我们直接对之前的 C++ 程序代码进行一些细节上的修改，具体代码如下。

```
fib.cc
// 引入 Emscripten 工具链用到的头文件
#include <emscripten/emscripten.h>

#ifdef __cplusplus
extern "C" {
#endif
// 使用 EMSCRIPTEN_KEEPALIVE 来标记需要导出的函数
int EMSCRIPTEN_KEEPALIVE fib (int x) {
    if (x < 2) {
        return 1;
    } else {
        return fib(x - 1) + fib(x - 2);
    }
}

#ifdef __cplusplus
}
#endif
```

接下来使用 Emscripten 工具链对源代码进行编译，命令如下。

```
sudo emcc -s WASM=1 -O3 -o fib.js fib.cc
```

最后，我们仍然需要创建一个 HTML 文件来整合 Emscripten 工具链生成的 Wasm 模块和用于 JavaScript Binding 的脚本文件。在调用 Wasm 模块中用于推导斐波那契数列的方法时，我们使用 `performance.now` 函数来计算整个过程所花费的时间。

```
<DOCTYPE html>
<html lang="en">
  <head>
<meta charset="UTF-8">
<title></title>
  </head>
  <body>
```

```
<script>
// 初始化一个名为 Module 的全局对象，Emscripten 生成的 JavaScript Binding 脚本会自动初始化该对象
var Module = {};
var script = document.createElement('script');

// 从远程加载 Wasm 模块，然后获取模块内的字节数据
fetch('fib.wasm').then(response => response.arrayBuffer()).then((bytes) => {
  // 将 Wasm 模块的主要数据填充到 Module 对象中
  Module.wasmBinary = bytes;
  // 异步加载用于初始化的 JavaScript Binding 脚本
  script.src = 'fib.js';
  document.body.appendChild(script);
});

// Module 初始化后运行此处代码
script.onload = function() {
  const fibonacciNumber = 45;
  // JavaScript Binding 脚本加载完成后，此时的 Module 对象已经初始化完毕
  let startTime = performance.now();
  // 调用 Wasm 模块暴露出的方法
  Module["asm"]["_fib"](fibonacciNumber);
  console.log(`${performance.now() - startTime} ms`);
}
</script>
</body>
</html>
```

同样的，我们通过统计图来展示之前的原生 C++、PNaCl、ASM.js、原生 JavaScript，以及本次基于 Wasm 的应用在运行时的总体耗时情况，如图 1-34 所示。

可以看到，基于原生 C++语言和 PNaCl 技术开发的应用其运行效率仍然位于整个性能排行榜的前两位。虽然 Wasm 应用的性能排于前两者之后，但其总体的性能表现实际上并不差，三种应用的运行耗时十分接近。总体上看，原生 C++、PNaCl 和 Wasm 应用的性能均遥遥领先于 ASM.js 应用，以及使用原生 JavaScript 开发的应用。由此可见，JavaScript 引擎在解析和优化 JavaScript 代码时花费的成本是如此之高。例如 V8 等 JavaScript 引擎在 JavaScript 代码处理效率上的性能优化仍然存在瓶颈和巨大的空间。

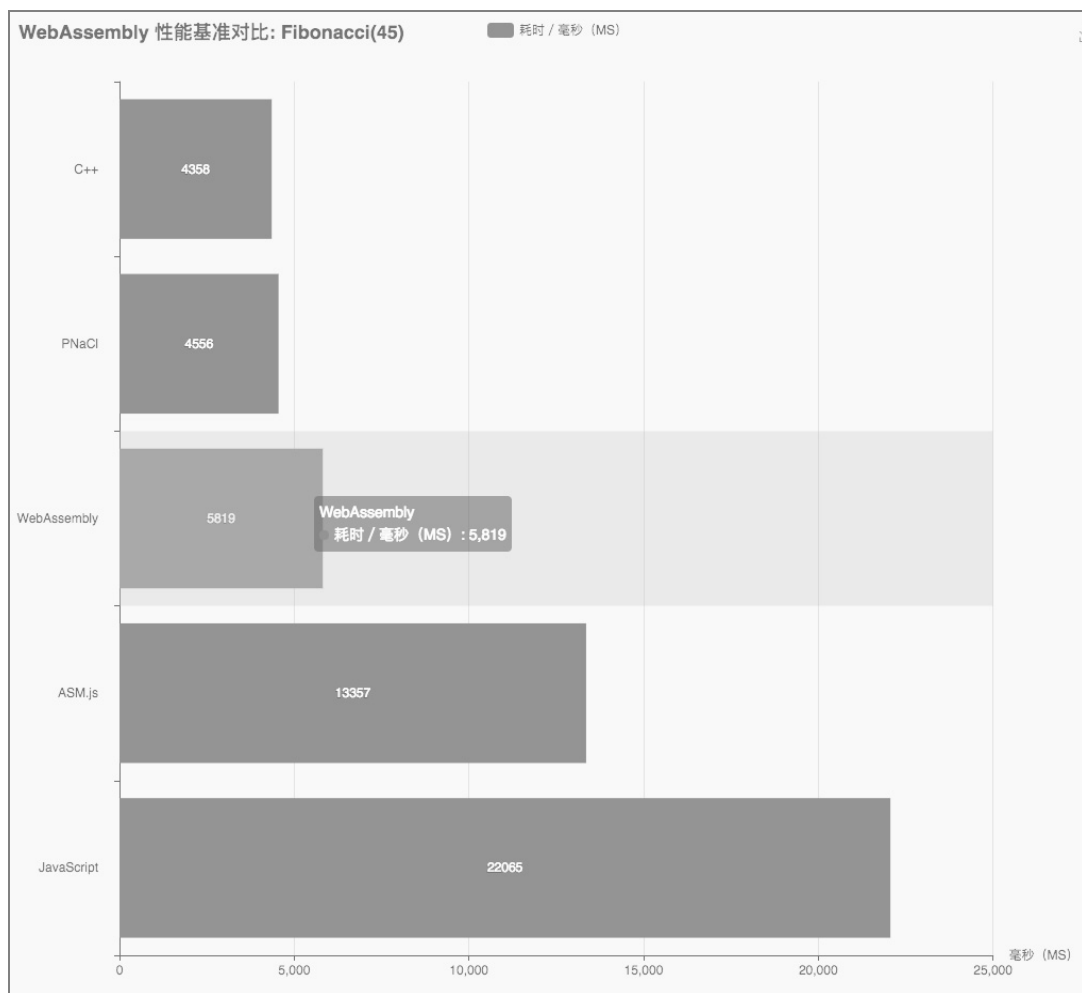


图1-34 WebAssembly性能基准对比

兼容性

通过上面的 Wasm 应用实例和基于基准测试的性能对比结果，我们大致了解了一个 Wasm 应用的具体开发流程，以及 Wasm 技术的基本实现原理。如果想要在生产环境中大范围使用基于 Wasm 技术开发的 Web 应用，一定要确保绝大多数的主流浏览器都能够支持这项技术。只有在这个基本前提下，才能保证基于该技术开发的应用可以覆盖到足够多的用户。那么，实际上各浏览器厂商对 Wasm 技术的支持程度如何呢？图 1-35 给出的是截至 2017 年年底，Wasm 技术标准在四大主流浏览器中的支持情况。

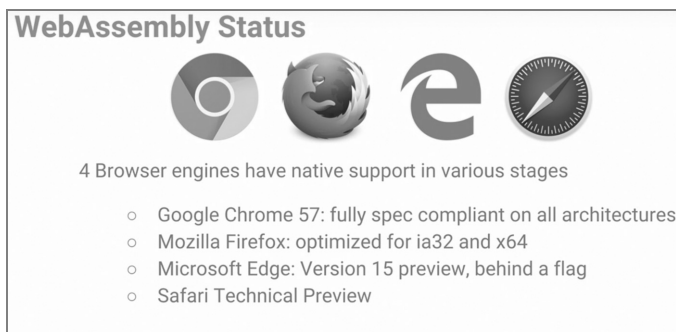


图1-35 截至2017年年底Wasm技术标准在四大主流浏览器中的支持情况

实际上，自 Wasm 技术开始出现在公众视野直到 2017 年 2 月底，包括 Chrome、Firefox、Microsoft Edge 和 Safari 在内的几大主流浏览器厂商宣布已经在其各自不同版本的浏览器中支持 Wasm 技术最小可用版本（MVP）标准中列出的所有特性。如果按照前面给出的各主流浏览器的市场占有份额来看，这四种主流浏览器各自的市场占有份额加在一起已经占到 93.21%，这标志着 Wasm 技术正式走上了从技术研究到规模化应用的转型道路。

不仅如此，2017 年 3 月谷歌 Chrome 官方团队宣布在其 Chrome 57 版本后的浏览器中，将会默认启用对 Wasm 技术标准的支持。用户升级浏览器后，不再需要通过“chrome://flags/”选项卡手动开启浏览器对 Wasm 的支持。同样的，2017 年 8 月微软也宣布将在其 EdgeHTML 16 排版引擎中默认开启对 Wasm 的支持。这些举措无疑都推动了 Wasm 技术的进一步发展。

PNACL 与 Wasm

从前面我们针对原生 C/C++、PNACL、Wasm 及其他几种技术或语言进行的性能基准测试结果中可以看到，在 Web 应用的性能方面，PNACL 和 Wasm 两者具有独特的优势。接下来，我们将详细对比这两种技术的特征，并总结出如下几点优劣比较。

- 自 Wasm 技术出现后，Chrome 团队官方宣布已经放弃对 PNACL 技术的后续开发和维护，官方已经在 PNACL 开发者文档首页中建议开发者将现有的基于 PNACL 技术开发的应用迁移到新的 Wasm 应用上。
- PNACL 应用开发流程复杂。即使完成一个最简单的 Web 应用也需要基于 Pepper API 进行专门的开发，上手难度较大。而 Wasm 应用可以基于原生的 C/C++ 标准库进行独立开发，即使需要使用到 Emscripten 工具链提供的一些底层 VM 独有的特性，其整个开发流程也是轻耦合，且十分方便和简单。

- PNaCl 并不支持对原生 DOM 对象的直接操作。而 Wasm 在其后续的标准迭代计划中将会引入操作原生 DOM 的相关标准，这无疑会极大拓宽 Wasm 技术在 Web 前端应用开发中的业务场景。
- PNaCl 存在兼容性问题。基于 PNaCl 技术开发的应用只能够运行在 Chrome 浏览器中，这无疑限制了对产品进行规模化分发的可能。而 Wasm 技术已经被四家主流浏览器厂商支持，不仅如此，那些基于 Webkit 或 V8 引擎开发的其他品牌浏览器也会逐渐对 Wasm 技术启用支持。这无疑是 Wasm 技术相对于 PNaCl 的一个巨大优势。

Wasm 技术的出现，不仅使 Web 浏览器运行高性能的计算密集型应用成为可能，而且传统的复杂 Web 前端应用存在的性能问题，相信在不久的将来也极有可能通过 Wasm 技术得以解决。

应用领域

基于各大浏览器已经实现的 Wasm 技术最小可用版本（MVP），现阶段我们能够实现哪些类型的应用呢？按照具体的运行环境可以将这些应用分成两类：一类是需要运行在 Web 浏览器上的 Wasm 应用；另一类是可以运行在非 Web 浏览器环境下的 Wasm 应用。

基于 Web 浏览器提供的 Wasm 技术，我们可以实现如下类型的应用。

- 图像/视频编辑器。
- 在线的休闲类小游戏，甚至是跨平台的大型网络游戏。
- 在线的点对点应用。
- 在线的流媒体应用。
- 在线的图片识别应用。
- 虚拟现实（AR）类应用。
- 开发者使用的在线编辑器、编译器及调试器。
- 虚拟机和平台模拟器。
- 远程桌面应用。
- VPN 应用。

- 加密服务类应用。

基于非 Web 平台提供的 Wasm 技术，有如下类型的应用。

- 基于 Wasm 的混合类移动应用。
- 基于 Wasm 的分布式多节点均衡计算类应用。
- 游戏分发类服务。

随着 Wasm 技术后续 Post-MVP 标准的逐渐确定和实现，相信 Wasm 技术的应用领域将会变得越来越广，逐渐从一些偏重功能性的密集计算类应用转向更为普遍的 Web 前端应用。

1.3.2 一路向前，WCG 与 WWG

在前面的章节中，我们为读者简单介绍了 Wasm 技术的基本原理、应用的开发流程，以及 Wasm 在应用性能、浏览器兼容性和可用领域等方面的内容。接下来，我们将从历史的角度着眼，探寻 WebAssembly 技术发展背后的“历史故事”。

发展历史

Wasm 技术第一次以官方名义对外界公布的时间是在 2015 年的 6 月 12 日。如图 1-36 所示，这一天 WCG (WebAssembly Community Group) 的提名主席 Jean-Francois Bastien (后面简称 JF) 在 WebAssembly 的官方 Github 仓库中发布了一则消息，该消息称 WCG 将会于 2015 年 6 月 17 日的太平洋时间 9 点钟将 Wasm 技术当前阶段的发展计划公之于众。WCG 成立于 2015 年 4 月，设立 W3C Community Group 的意图在于为开发人员、设计人员，以及热衷于为 Web 标准做出贡献的人员提供一个可以进行讨论和发表想法的地方。这些小组都是由社区提出并开始运营的，WCG 便是其中的一个。

2015 年 6 月 17 日，WCG 以官方名义将 Wasm 技术的发展规划公之于众后，Wasm 便开始进入了一个飞速发展的阶段。WCG 的核心成员主要由一群来自苹果、谷歌、微软及 Mozilla 等互联网巨头公司的顶尖工程师组成，这些工程师会定期对 Wasm 技术的标准进行调研和讨论。所有与 Wasm 技术相关的标准草案，以及未来的发展路线都由他们共同制定。当然，你也可以选择加入 WCG，为 Wasm 标准的未来发展提出自己的建议和意见。WCG 免费向公众开发者开放，你只需要注册成为 W3C 官方成员即可快速加入。

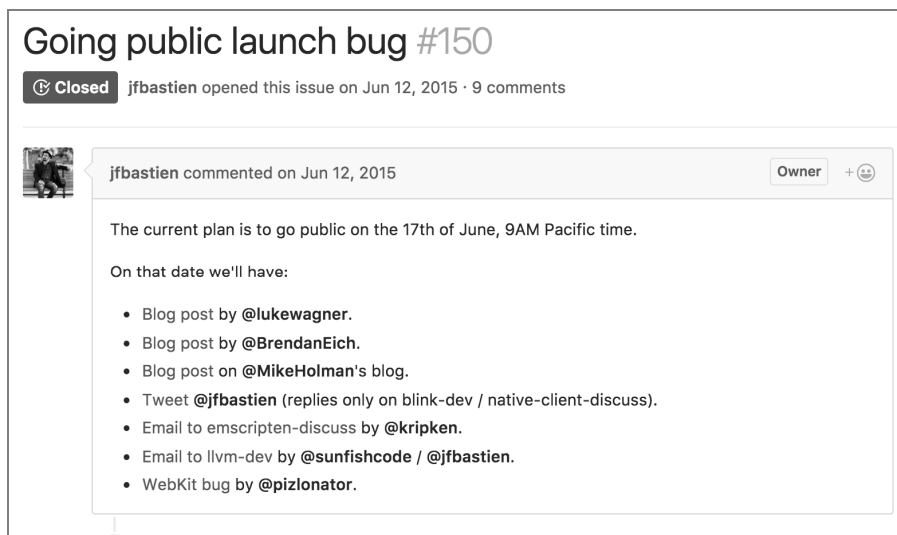


图1-36 JF在Github上讨论Wasm标准的公开计划

在 WCG 官方首次对外宣布 Wasm 技术标准和发展计划 9 个月后的 2016 年 3 月, 来自 Mozilla 的 Luke Wagner 在其博客中提到了 Wasm 技术的最新发展情况——在这篇博客发布之时, 已经有多家主流浏览器厂商在其浏览器中实现了一些实验性的 Wasm 技术标准。同时, WCG 官方也已经在 Wasm 的标准制定上取得了多项进展。四家主流浏览器厂商已经在其各自的浏览器中实现了 Wasm 技术标准原型版本。WCG 官方提供的一个基于 Unity3D 实现的用于测试的 Demo 版游戏已经可以被成功地编译到对应的 Wasm 版本, 并顺利地在这些浏览器上运行。至此, 四大主流浏览器厂商开始在 Wasm 技术标准的跟进上保持同步的节奏, 这也是 Wasm 技术发展中的一个里程碑时刻。

2016 年 8 月, Wasm 开始了漫长的“Browser Preview”阶段。在这一阶段, Wasm 技术会以实验性功能的形式在各大浏览器中提供给开发者使用。WCG 官方宣布此时已经在多个不同类型的 Web 浏览器中实现了一套统一的 Wasm 模块文件标准, 并且可以让开发者使用并反馈建议和意见。

2017 年 2 月, 经过开源社区众多开发者的投票, Wasm 的官方 LOGO 从众多的设计作品中被最终确定下来。

众多参与投票的 LOGO 作品如图 1-37 所示。

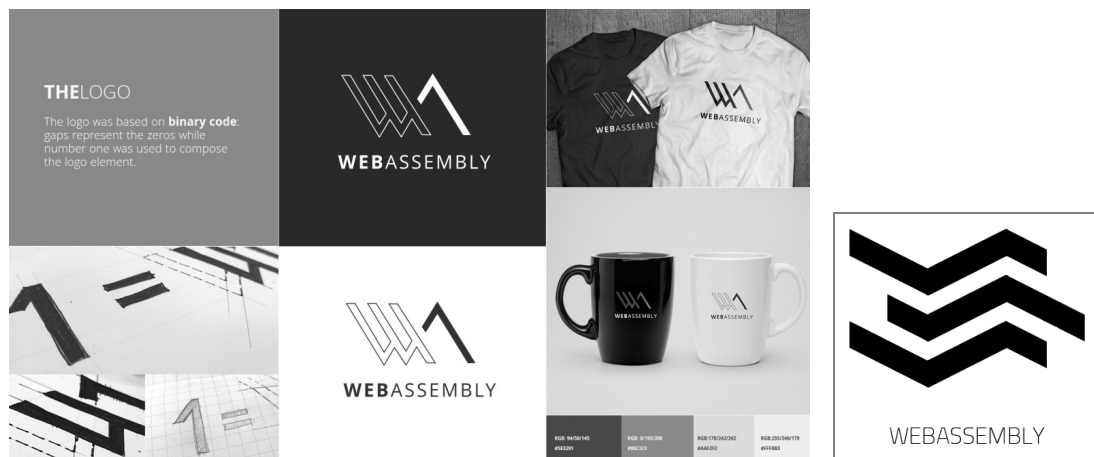


图1-37 参与投票的LOGO作品

最终被确定的 LOGO 设计方案如图 1-38 所示。

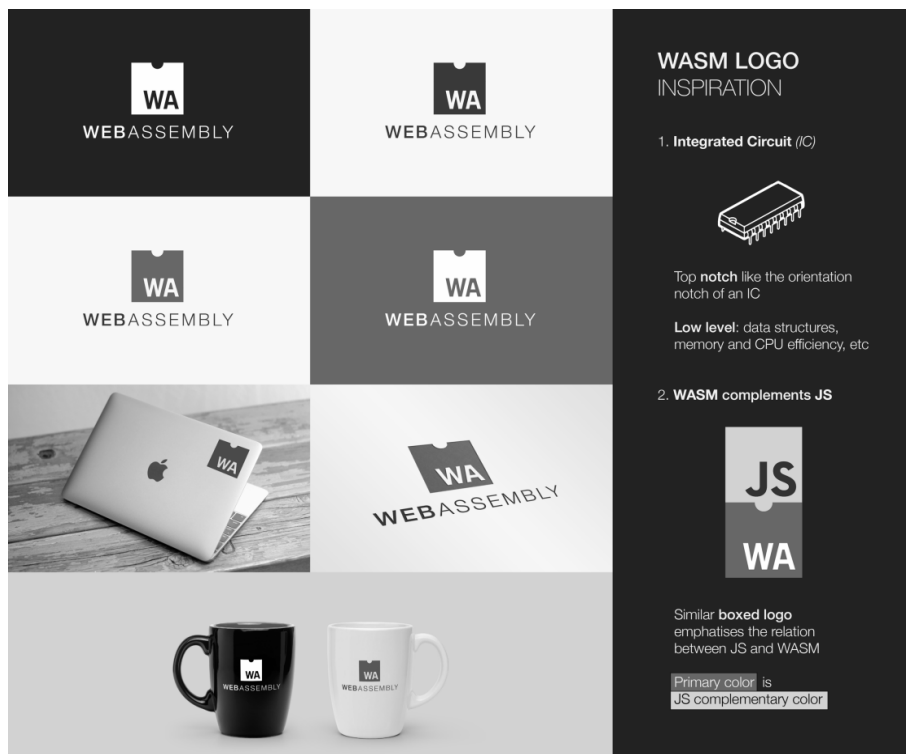


图1-38 最终确定的LOGO设计方案

2017 年 2 月 28 日，四大浏览器厂商在 Wasm 技术的 MVP 标准实现上达成共识。这意味着 Wasm 在其最小可用版本（MVP）上的“Browser Preview”阶段已经结束。浏览器可以正式以默认启用的方式来支持 Wasm 技术，开发者不再需要通过手动打开浏览器实验性功能或下载技术预览版本的方式来开启浏览器对 Wasm 特性的支持。

2017 年 8 月 3 日，W3C WebAssembly Working Group（WWG）成立。从 WCG 到 WWG 的转变标志着 Wasm 技术将同 HTTP 等常用的 Web 技术一样，成为 W3C 标准技术体系的一部分。W3C 工作组主要负责制定未来几年内 Wasm 标准的开发计划，并继续迭代当前社区组（WCG）正在实现的特性。

W3C WWG

WWG 的主要任务是负责推动 Wasm 技术的标准迭代，以及约束在各种不同类型浏览器上实现的 Wasm 标准能够保持对用户统一的响应行为。在 WWG 的官方宪章说明中我们可以看到 Wasm 技术的整个发展时间线，WWG 小组计划通过两年的时间来推动 Wasm 技术的整体发展。如图 1-39 所示，整个 Wasm 技术的发展过程会被分为三个阶段，在每个阶段中都对应有一次标准草案的变更和重新发布。每一次标准草案的变更都是根据之前 6 个月内 WWG 和 WCG 小组对当前标准的讨论结果最终制定的，这些对标准草案的变更结果会使 Wasm 技术在平台兼容性、底层运行效率，以及各种技术细节的优化上都得到提升。

2.3 Timeline
• Q4 2017: First Public Working Draft of WebAssembly v.1 • Q2 2018: Recommendation of WebAssembly v.1 • Q2 2018: First Public Working Draft of WebAssembly v.2 • Q4 2018: Recommendation of WebAssembly v.2 • Q4 2018: First Public Working Draft of WebAssembly v.3 • Q2 2019: Recommendation of WebAssembly v.3

图1-39 Wasm技术的发展规划时间表

WCG 会通过多种方式定期举行关于 Wasm 标准相关议题的讨论会。比如每月举行一到两次的短时在线视频讨论会，以及每年在各大互联网巨头总部举行长达数日的线下研讨会议。你可以通过关注 WebAssembly 官方 Github 仓库或加入 WCG 并订阅邮件列表的方式来获取这些会议的安排时间表。WWG 会以完全开放的态度来接收和采纳开发者对 Wasm 标准提出的建议和意见，因此你可以通过参加这些定期举行的线上/线下研讨会来与 Wasm 核心团队的成员进行面对面的交流。如图 1-40 所示的是笔者参与某次 WCG 线上讨论会的视频截图。

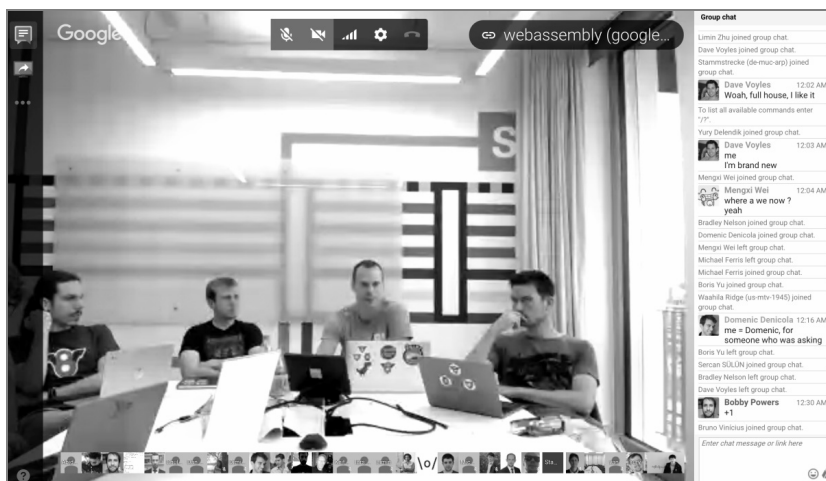


图1-40 笔者参与某次WCG线上讨论会的视频截图

总结：本章主要介绍了 JavaScript 语言与 JS 引擎存在的问题、ASM.js 与 NaCl/PNaCl 等新技术做出的尝试及存在的问题。WebAssembly 技术一路走来在各个历史阶段的发展历程，并通过几个简单的实例介绍了构建 Wasm 应用的基本方法及相关实现细节。在下一章中，我们将开始深入讨论 Wasm 标准的底层技术细节，如 Wasm 模块的二进制编码方式、独立于线性内存的调用堆栈结构，以及模块的二进制编码与内存结构等相关内容。

第 2 章

WebAssembly 核心原理（基于 MVP 标准）

通过第 1 章的介绍，相信大家已经对 WebAssembly 技术的发展历程，以及基于 Emscripten 工具链构建 Wasm 应用的基本流程有了一个初步的认识。在本章中，我们将深入到 WebAssembly 标准的内部，来探究其从上层标准 JavaScript 接口、模块内存结构一直到底层堆栈机模型和模块二进制编码格式等核心部分的设计原理。

2.1 应用与标准 Web 接口

在前面的内容中，我们曾简单介绍了如何通过 Emscripten 工具链来快速构建一个完整的 Wasm 应用。Emscripten 工具链为我们自动生成了可以方便地在 Web 浏览器中连接并初始化 Wasm 模块的“JavaScript Binding”脚本。在该脚本文件中，Emscripten 通过调用在 Wasm 标准中定义的相关上层 JavaScript 接口来加载及初始化 Wasm 模块实例，并将模块内的方法导出。本节我们将介绍这些在 MVP（最小可用产品）版本标准中定义的与 WebAssembly 应用层相关的 JavaScript 接口，以及它们的具体用法。

2.1.1 编译与初始化

为了能够在 Web 浏览器中正常地加载 Wasm 模块并使用其暴露出的相关功能，Wasm 官方开发团队在设计和实现其 MVP 标准时，也设计了一套可以在 Web 浏览器中通过 JavaScript 代码处理和操作 Wasm 模块资源的通用上层 JavaScript API 接口。这些标准接口已经被内置到了 JavaScript 标准库中，因此我们可以直接在 Web 浏览器中通过 JavaScript 调用这些标准接口来加

载 Wasm 模块并处理模块的相关资源。

首先，我们通过一个使用标准 JavaScript API 加载 Wasm 模块的例子，来了解在 Web 浏览器中应该如何使用“标准程序”来加载和使用一个 Wasm 模块。在这个例子中，我们会使用到大部分常用的 WebAssembly JavaScript 标准 API，关于这些 API 的详细使用方法会在本节后面的内容中进行介绍。

该例子中的 Wasm 模块会向 JavaScript 环境中暴露一个方法，该方法用来对特定的 JavaScript 数组进行正向排序。与第1章在 Wasm 模块中使用的排序方法不同的是，在这里当该 Wasm 模块被加载到内存并被初始化时，这个需要排序的数组结构便已经生成了，数组的大小及其在内存中的首地址也已经确定。我们唯一需要做的便是从 JavaScript 环境向该数组中填充一些数据，然后再通过 Wasm 模块暴露出的方法对内存中这个特定数组中的元素进行排序处理。

对于这种不需要依赖 Emscripten 工具链运行时环境来运行的 Wasm 模块，我们称之为 Standalone 模块。这种类型的模块在 Web 浏览器中运行时不需要使用特定的初始化脚本（比如 JavaScript Binding 脚本），只需要通过调用浏览器提供的内置于 JavaScript 标准库中的标准 WebAssembly API 接口便可以完成 Wasm 模块的加载及初始化操作。为了快速地从 C/C++ 源代码编译出一个 Standalone 类型的模块，这里我们通过 WasmFiddle 平台来快速地进行构建。如图 2-1 所示，便是 WasmFiddle 平台的 WebAssembly 在线开发、编译和运行环境。

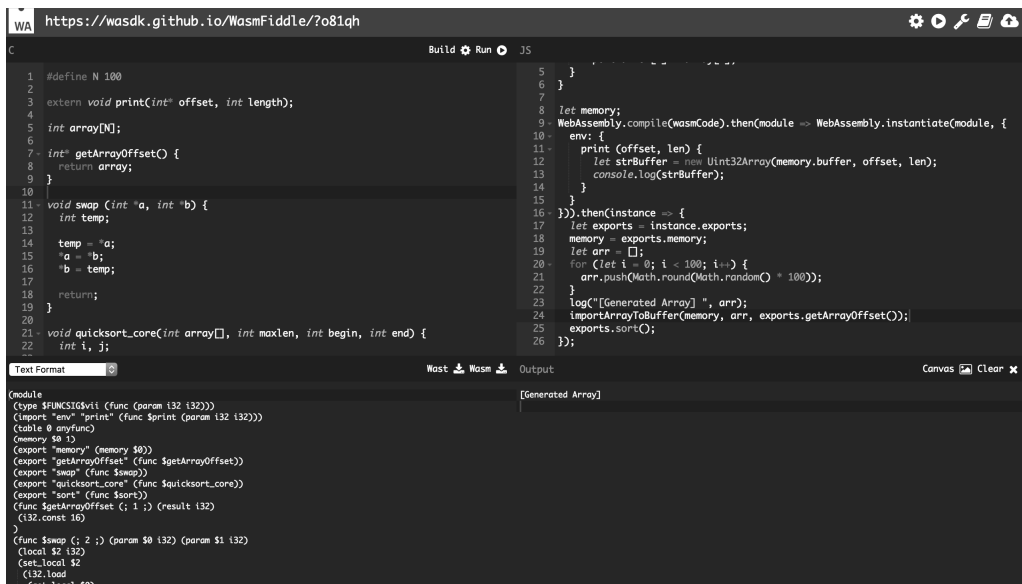


图2-1 WasmFiddle平台的WebAssembly在线开发、编译和运行环境

首先我们给出模块的 C/C++源代码。由于不需要借助 Emscripten 工具链来编译模块，因此在这段源代码中并不会涉及与 Emscripten 工具链提供的预置宏以及头文件相关的代码。

```
standard_sort.cc
```

```
// 定义待排序数组的大小
#define N 10

// 定义将从 JavaScript 环境导入的函数
extern void print(int* offset, int length);
// 声明用于排序的全局数组，在模块初始化时便已在内存中生成
int array[N];

// 用于在 JavaScript 环境中获取待排序数组首地址的方法
int* getArrayOffset() {
    return array;
}

void swap (int *a, int *b) {
    int temp;

    temp = *a;
    *a = *b;
    *b = temp;

    return;
}

// 核心的快速排序方法
void quicksort_core(int array[], int maxlen, int begin, int end) {
    int i, j;

    if(begin < end) {
        i = begin + 1;
        j = end;

        while(i < j){
            if(array[i] > array[begin]) {
                swap(&array[i], &array[j]);
                j--;
            } else {
                i++;
            }
        }
    }
}
```



```

    }
}

if(array[i] >= array[begin]) {
    i--;
}

swap(&array[begin], &array[i]);

quicksort_core(array, maxlen, begin, i);
quicksort_core(array, maxlen, j, end);
}
}
// 用于在 JavaScript 环境中调用的主排序的方法
void sort () {
    quicksort_core(array, N, 0, N - 1);
    // 调用从 JavaScript 环境导入的“打印”方法并将数组作为参数传入
    print(array, N);
}

```

在 WasmFiddle 平台操作界面左上角的 C/C++源代码区域输入上述给出的代码，然后点击该区域上方工具栏中的“Build”按钮，即可启动平台对 C/C++源代码的编译过程。当编译完成后，工具栏中的“齿轮”图标会停止转动。此时我们便可以通过界面下方工具栏中的“Wasm”按钮将已经编译好的 Wasm 模块文件下载到本地。为了能够更好地演示如何在本地环境中应用一个 Standalone 模块的完整流程，我们不会直接使用 WasmFiddle 平台右侧的 JavaScript 脚本区域来执行 JavaScript 代码。下面给出用于加载和调用 Wasm 模块的 HTML 文件代码。同样的，为了方便起见，我们会将 HTML、CSS 和 JavaScript 三种类型的代码写在同一个 HTML 文件中。

```

standard_sort.html
<!DOCTYPE html>
<html>
<head>
    <title>使用标准程序加载 Wasm 模块</title>
</head>
<body>
    <p><span>排序前:</span><span class="sequence-before"></span></p>
    <p><span>排序后:</span><span class="sequence"></span></p>
<script>
// 该方法用于从 JavaScript 环境向指定的共享堆内存填充数据

```

```

function importArrayToBuffer (memory, array, offset) {
  const importBuffer = new Uint32Array(memory.buffer, offset, array.length);
  for (let i = 0; i < array.length; i++) {
    importBuffer[i] = array[i];
  }
}

// 从远程加载一个 Wasm 模块，并将该模块中的内容转换成二进制数据
let startTime = performance.now();
fetch('original.wasm').then(response => response.arrayBuffer()).then((bytes) => {
  let memory;
  // 通过浏览器提供的标准 WebAssembly 接口来编译和初始化一个 Wasm 模块
  WebAssembly.compile(bytes).then(module => WebAssembly.instantiate(module, {
    env: {
      // 需要导入到模块中的 JavaScript 函数体
      print (offset, len) {
        let strBuffer = new Uint32Array(memory.buffer, offset, len);
        document.querySelector('.sequence').innerText =
JSON.stringify(Object.values(strBuffer));
      }
    }
  })).then(instance => {
    // 输出下载、编译及实例化 Wasm 模块花费的时间
    console.log(performance.now() - startTime);
    // 取出从 wasm 模块中导出的函数
    let exports = instance.exports;
    // 获得该 Wasm 模块实例使用的堆内存对象
    memory = exports.memory;
    let arr = [];
    // 生成一个包含有 10 个元素的整型数组
    for (let i = 0; i < 10; i++) {
      arr.push(Math.round(Math.random() * 10));
    }
    document.querySelector('.sequence-before').innerText = JSON.stringify(arr);
    // 将整型数组内的元素依次填充到指定的内存段中，即填充到 wasm 模块初始化时生成的数组中
    importArrayToBuffer(memory, arr, exports.getArrayOffset());
    // 调用 Wasm 模块暴露出的排序函数
    exports.sort();
  });
});

```

```
</script>
</body>
</html>
```

至此，一个通过 WasmFiddle 平台构建的 Wasm 模块，以及用于加载该模块的 JavaScript 脚本代码及 HTML 页面已经全部准备完毕。为了能够在本地环境中运行该 Wasm 应用，我们仍然需要启动一个小型的本地 HTTP 服务器来提供服务。服务器启动后，我们可以在浏览器中访问之前创建的 HTML 文件。当 Wasm 应用启动并成功运行完毕后，可以在浏览器中看到如图 2-2 所示的页面内容。这里我们将经过 Wasm 模块排序前后的数组内容打印在了浏览器中。

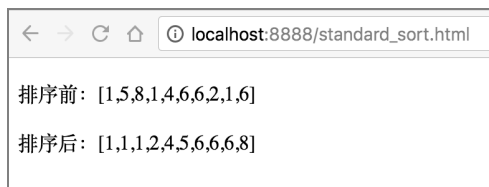


图2-2 上述Wasm应用在浏览器中的实际运行效果

可以看到，Wasm 模块已经成功被加载和初始化，同时也完成了对指定数组元素的正向排序过程。下面我们来看一下在上述 JavaScript 脚本中出现两个分别用于编译和初始化 Wasm 模块的标准 JavaScript 接口方法，即 `WebAssembly.compile()` 和 `WebAssembly.instantiate()`。

WebAssembly.compile(bufferSource)

该方法接收一段标准的二进制格式的 WebAssembly 模块代码，编译并返回一个对应于模块的 `WebAssembly.Module` 对象。`WebAssembly.compile` 方法接收的模块代码必须是以类型数组 `TypedArray` 或二进制缓冲区 `ArrayBuffer` 结构表示的二进制 Wasm 模块代码。该方法会对这些二进制形式的代码进行编译处理，然后返回一个可以解析为 `WebAssembly.Module` 对象实例的 `Promise` 结构。因此，我们说 `WebAssembly.compile` 方法仅用来对 Wasm 模块进行编译处理，而并不会对模块进行实例化操作。

在 `WebAssembly.Module` 对象中实际上包含有一些已经由浏览器编译好的无状态的 Wasm 模块代码。除了通过 `WebAssembly.compile` 方法来创建 `WebAssembly.Module` 对象实例，我们还可以直接通过 `WebAssembly.Module` 的构造函数来构造对象实例。这两种方法的唯一区别是，前者是以异步方式进行的，而后者则是以同步方式进行的。因此，在绝大部分情况下，我们更建议开发者使用 `WebAssembly.compile` 方法以异步返回 `Promise` 对象的方式来创建 `WebAssembly.Module` 对象实例。

不仅如此，`WebAssembly.Module` 对象本身也是可以“转移”的。这意味我们可以将该对

象直接存储到浏览器的 IndexedDB 数据库中，并在需要的时候再读取出来直接使用。另外，也可以在浏览器的主线程和 Worker 线程之间互相传递 WebAssembly.Module 对象，将一些可能会消耗大量计算资源的模块方法调用放在 Worker 线程中进行处理，进而防止阻塞主线程。

实际上，将 WebAssembly.Module 对象实例在各环境中转移的过程，就是将该对象的二进制源进行克隆并在新的目标环境中重新编译使用的过程。而浏览器在重新编译的过程中会优先尝试使用其内部在之前过程中已经编译好的 WebAssembly.Module 副本对象，这使得当我们在新环境中使用这些 WebAssembly.Module 对象时，在大部分情况下是不需要进行再编译的。这种对象的转移方式也被称为“结构化克隆”。

WebAssembly.Module 的构造函数除可以用于构造对象实例以外，在该构造函数中还提供了一些静态方法来获取 Wasm 模块对象的内部信息。

WebAssembly.Module.customSections(module, sectionName)

该方法可以返回指定 Wasm 模块实例中某个自定义段的内容副本。每一个标准的 Wasm 模块都是由许多“标准段”和“自定义段”组成的，在每一个段结构中都包含有用于描述该模块的不同内容。标准段由 Wasm 官方标准定义，它们有着固定的格式和内容；而自定义段的内容则不受限制，可以由开发者自行定义和使用。浏览器在初始化一个 Wasm 模块时会验证模块标准段中的内容是否符合标准规范的定义，如果不符合规范则会抛出异常。

我们可以将上一个 Wasm 实例中的 JavaScript 脚本代码稍做修改，并尝试使用 WebAssembly.Module.customSections 方法来查看一个 Wasm 模块的自定义段内容。该方法接收两个参数，第一个参数为已经编译好的 WebAssembly.Module 对象实例；第二个参数为想要获取的该模块的某个段内容副本的“段”名称。我们可以使用如下给出的 JavaScript 代码来查看在该 Wasm 模块内部位于自定义段的二进制数据内容。

```
...
fetch('original.wasm').then(response => response.arrayBuffer()).then(bytes => {
  // 通过浏览器提供的 WebAssembly API 接口来加载一个 Wasm 模块
  WebAssembly.compile(bytes).then(module => {
    // 查看该 Wasm 模块实例中名为 customized 的自定义段内容
    console.log(WebAssembly.Module.customSections(module, 'customized'));
  });
});
...
```

这里我们只给出了主要更改部分的 JavaScript 脚本代码，如果需要在本地环境中进行测试，

可以参考之前给出的完整实例并将其他部分的代码补全。当在浏览器中运行上述 JavaScript 脚本代码时，会发现 `WebAssembly.Module.customSections` 方法返回了一个空数组，而这是一个正确的运行结果。由于该方法只会返回 Wasm 模块中自定义段的内容，而我们并没有在之前编译的 Wasm 模块中指定名为“customized”的自定义段，因此返回的结果便是一个长度为 0 的空数组，即函数没有返回任何具有内容的二进制缓冲区数据。

`WebAssembly.Module.exports(module)`

该方法会返回从 Wasm 模块实例导出到 JavaScript 环境中的所有对象的描述信息。Wasm 模块可以向 JavaScript 环境中导出一些具有不同功能和种类的 JavaScript 对象，现阶段最常见的就是从模块导出的，代表模块核心功能的处理函数。可以用如下代码来查看模块导出对象的描述信息。

```
...
fetch('original.wasm').then(response => response.arrayBuffer()).then(bytes => {
  WebAssembly.compile(bytes).then(module => {
    // 查看从该 Wasm 模块实例导出到 JavaScript 环境中的对象信息
    console.log(WebAssembly.Module.exports(module));
  });
});
...
```

运行上述代码，我们可以在浏览器控制台中看到如图 2-3 所示的结果。

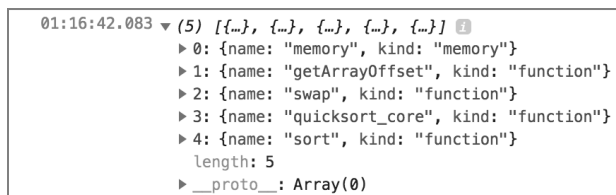


图2-3 使用`WebAssembly.Module.exports`方法查看模块导出的外部对象信息

可以看到，`WebAssembly.Module.exports` 方法返回了从 Wasm 模块内部导出到 JavaScript 环境中的所有对象信息。该方法返回的数组结构中包含了多条记录，每一条记录都对应着一个导出对象的完整描述信息。这些记录标识了每个导出对象的具体类型及名称。在 MVP 标准中规定，一个模块对象实例能够向 JavaScript 环境中导出的资源对象类型可以为 `function`、`table`、`memory` 和 `global` 中的任意一种。

`WebAssembly.Module.imports(module)`

该方法会列出 Wasm 模块在进行实例化时需要从 JavaScript 环境中导入的对象信息。为了

能够更灵活地在 JavaScript 环境与 Wasm 模块对象实例进行交互，我们可以直接从 JavaScript 环境向正在进行实例化的 Wasm 模块对象实例传递一些数据。在 MVP 标准中，同上面的导出数据一样，这些模块导入数据也只能为 function、table、memory 和 global 类型中的任意一种。

比如在之前的 Wasm 排序实践代码中，我们向正在进行实例化的 Wasm 模块传递了一个名为“print”的 JavaScript 函数（对应 function 类型）。而为了能够在 Wasm 模块内部正常地使用这个 JavaScript 函数，便需要在 C/C++代码中定义该函数的原型。通过函数原型，模块可以告知 C/C++编译器及浏览器引擎应该如何为该函数分配资源。因此，只有当传入模块的函数与其在 Wasm 模块内事先定义好的函数原型完全匹配后，浏览器才能够正确地进行函数调用。接下来，我们还是通过一段代码来查看上述 Wasm 模块中有哪些已经声明好的函数原型，以及还需要从 JavaScript 环境中导入的其他对象信息。

```
...
fetch('original.wasm').then(response => response.arrayBuffer()).then(bytes => {
  WebAssembly.compile(bytes).then(module => {
    // 查看从该 Wasm 模块实例导出到 JavaScript 环境中的对象信息
    console.log(WebAssembly.Module.imports(module));
  });
});
```

在浏览器中运行上述代码，运行结果如图 2-4 所示。

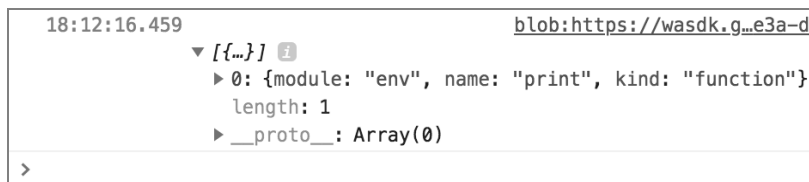


图2-4 使用WebAssembly.Module.imports查看需要导入到模块的外部对象

可以看到，如果要对该 Wasm 模块进行初始化操作，则需要将原生 JavaScript 函数即 print 函数放到一个名为“env”的 JavaScript 对象中，然后再将 env 对象整体作为参数通过 WebAssembly.instantiate 标准接口传递给正在初始化的 Wasm 模块实例。而这里提到的 WebAssembly.instantiate 方法，便是整个 WebAssembly 标准 JavaScript 接口中用于编译和实例化 Wasm 模块的核心方法。

WebAssembly.instantiate()

该方法是用来编译和实例化 Wasm 模块的核心方法。它有如下两种重载形式。

WebAssembly.instantiate(bufferSource, importObject)

通过这种重载形式，我们可以直接通过含有 Wasm 模块二进制数据的 `TypedArray` 类型数组或 `ArrayBuffer` 二进制缓冲区，来编译并实例化一个标准的 Wasm 模块对象实例。

在这种重载形式下，该方法一共接收两个参数。第一个参数 `bufferSource` 表示包含有 Wasm 模块二进制数据的 `TypedArray` 或 `ArrayBuffer` 数据结构；第二个参数 `importObject` 是一个原生的 JavaScript 对象，在该对象中包含有需要在 Wasm 模块的实例化过程中导入到模块实例中的对象。`importObject` 参数中的导入对象需要按照“所在模块-对象实体”的形式进行组织。比如在本章最开始的实践项目中，用于初始化 Wasm 模块的 `WebAssembly.instantiate` 接口的第二个参数是按照如下形式进行组织和传入的。

```
...
{
  env: {
    print (offset, len) {
      let strBuffer = new Uint32Array(memory.buffer, offset, len);
      // console.log(strBuffer);
    }
  }
}
...
```

在整个 `importObject` 参数代表的对象结构中，其顶层被划分为多个不同的模块命名空间。比如在这里我们指定了一个名为“env”的模块命名空间。接下来，我们便可以在这个顶层模块的命名空间对象结构中添加需要传递给模块的各类对象（前述的四种类型）。比如常用的对应于 `function` 对象类型的原生 JavaScript 函数、对应于 `memory` 对象类型的 `WebAssembly.Memory` 堆内存段等。在上述代码中，我们在 `env` 模块命名空间中传入了名为“print”的 JavaScript 函数，即一个 `function` 对象类型的实体。

当 `WebAssembly.instantiate` 方法执行完毕后，会返回一个包含了结果对象的 `Promise` 对象，而在这个结果对象中又包含了两个不同的字段。第一个字段是一个已经被编译好的 `WebAssembly.Module` 对象实例；第二个字段是一个已经实例化的 `WebAssembly.Instance` 对象。我们之前介绍过，在 `WebAssembly.Module` 对象中主要包含有一些已经由浏览器编译好的无状态的 Wasm 模块代码，相对的，`WebAssembly.Instance` 对象则表示一个有状态的、已经被实例化的 `WebAssembly.Module` 对象。我们可以直接通过 `WebAssembly.Instance` 对象实例来使用 Wasm 模块导出的所有函数及其他对象资源。

除此之外, `WebAssembly.Instance` 本身也可以作为一个构造方法来使用。通过该构造方法, 我们可以将一个无状态的 `WebAssembly.Module` 对象实例化成一个有状态的 `WebAssembly.Instance` 对象。这个构造方法的功能与 `WebAssembly.instantiate` 方法保持一致。但需要注意的是, 所有与 `WebAssembly` 相关的构造方法都是以同步方式进行的。下面我们会将本章开头实践项目中的部分代码分别使用 `WebAssembly.instantiate` 的第一种重载形式和 `WebAssembly.Instance` 构造方法进行改写。首先是 `WebAssembly.instantiate` 的第一种重载形式, 代码如下。

```
...
let memory;
WebAssembly.instantiate(wasmCode, {
  env: {
    print (offset, len) {
      let strBuffer = new Uint32Array(memory.buffer, offset, len);
      console.log(strBuffer);
    }
  }
})
// Promise 对象在解析完成后会返回一个结果对象, 其中包含了无状态的 WebAssembly.Module 对象实例和有状
// 态的、经过实例化的 WebAssembly.Instance 对象实例
}).then(resultObject => {
  // 从 WebAssembly.Instance 对象实例中获取 Wasm 模块导出到 JavaScript 环境中的所有对象
  let exports = resultObject.instance.exports;
  // 获取从 Wasm 模块导出的堆内存对象
  memory = exports.memory;
  let arr = [];
  for (let i = 0; i < 10; i++) {
    arr.push(Math.round(Math.random() * 10));
  }
  console.log("[Generated Array] ", arr);
  importArrayToBuffer(memory, arr, exports.getArrayOffset());
  exports.sort();
})
...
```

接下来是 `WebAssembly.Instance` 构造方法, 代码如下。

```
...
let memory;
WebAssembly.compile(wasmCode).then(module =>
  // WebAssembly.Instance 构造方法同步实例化一个 WebAssembly.Module 对象。这里的
  // WebAssembly.Module 是通过 WebAssembly.compile 方法编译而来的
```



```

WebAssembly.Instance(module, {
  env: {
    print (offset, len) {
      let strBuffer = new Uint32Array(memory.buffer, offset, len);
      console.log(strBuffer);
    }
  }
})
// WebAssembly.Instance 构造方法只返回实例化后的 WebAssembly.Instance 对象
)).then(instance => {
  let exports = instance.exports;
  memory = exports.memory;
  let arr = [];
  for (let i = 0; i < 10; i++) {
    arr.push(Math.round(Math.random() * 10));
  }
  console.log("[Generated Array] ", arr);
  importArrayToBuffer(memory, arr, exports.getArrayOffset());
  exports.sort();
});
...

```

WebAssembly.instantiate(module, importObject)

`WebAssembly.instantiate` 方法的第二种重载形式在返回值类型上与第一种没有太大的区别。但这里需要将传入的第一个参数从 `TypedArray` 和 `ArrayBuffer` 这些以二进制形式数据描述的 Wasm 模块内容替换成对应的 `WebAssembly.Module` 对象，该对象实例可以通过 `WebAssembly.Module` 的构造方法或 `WebAssembly.compile` 方法直接从 Wasm 模块数据编译而来。在本章开头的实践例子中，我们便是通过这种方式对 Wasm 模块进行实例化的。

至此，我们已经介绍了几种用于编译和实例化 Wasm 模块的标准 JavaScript 接口方法。如果你擅于总结，就会发现无论通过哪种方式来编译和实例化 Wasm 模块，都需要首先通过 `FETCH API` 或普通的 `AJAX` 请求从远程下载 Wasm 模块文件。而且只有当 Wasm 模块文件的内容被全部下载到浏览器的内存中时，我们才能开始进行后续模块编译和实例化操作。对于一个仅提供了简单功能的 Wasm 模块，模块文件的下载、编译和实例化过程通常在几十到几百毫秒的时间内完成。但是对于一些使用 Wasm 技术构建的大型应用或框架（比如游戏、区块链应用及前端框架等），其对应模块的内部包含了大量的逻辑运算和数据处理流程，模块文件在下载、编译和实例化过程中可能会消耗大量的时间，进而影响了应用的实际用户体验。

为了能够进一步压缩 Wasm 模块从网络加载到完成实例化过程花费的时间，WebAssembly 团队在现有的 Web 标准接口之上又新增了两个可以用于直接从底层数据流源来编译和实例化 Wasm 模块的 JavaScript 方法。通过使用这两个方法，我们可以让 Wasm 模块的下载、编译及实例化过程同步地进行。

WebAssembly.compileStreaming(source)

该方法可以直接从浏览器底层的数据流源来编译 WebAssembly.Module 对象实例。在通常情况下，我们会直接将 FETCH API 中的 Response 对象作为统一的数据流源。因此在调用该方法时，可以直接传入一个 Response 对象，或者传入一个可以被解析为 Response 对象的 Promise 对象。

Response 对象是在 FETCH API 中定义的请求响应对象。在该对象中，以底层二进制数据流的方式保存着从远程服务器加载的数据信息。当我们调用 FETCH API 中用于从远程地址加载数据的核心方法 WindowOrWorkerGlobalScope.fetch 时，该方法便会将这些从远程位置获取到的数据以“包含 Response 对象的 Promise 对象”的形式返回给浏览器。因此，我们便可以直接在浏览器的全局环境中通过调用 fetch 方法来获得一个 Response 对象。接下来，我们将本章开头项目实例中的部分代码通过 WebAssembly.compileStreaming 方法进行改写，代码如下。

```
...
let startTime = performance.now()
// 通过 WebAssembly.compileStreaming 方法直接从底层数据流源编译一个 WebAssembly.Module 对象实例
WebAssembly.compileStreaming(fetch('original.wasm'))
.then(module => WebAssembly.instantiate(module, {
  env: {
    // 需要导入到模块中的 JavaScript 函数
    print (offset, len) {
      let strBuffer = new Uint32Array(memory.buffer, offset, len);
      document.querySelector('.sequence').innerText = JSON.stringify(Object.values
(strBuffer));
    }
  }
})).then(resultObject => {
  // 打印 Wasm 模块从远程服务器加载到完成实例化所花费的时间
  console.log(performance.now() - startTime);
  console.log(resultObject);
});
...
```

这里需要注意的是，从远程服务器加载的 Wasm 模块文件只有在其 HTTP 响应结果中被标识为“application/wasm”类型，才可以被 `WebAssembly.instantiateStreaming` 方法正确地编译和处理。因此，我们需要服务器程序在接收到获取 Wasm 模块文件的请求后，在对应资源的响应头中将其 MIME 类型标记为“application/wasm”。MIME 类型的存在是为了能够以一种标准化的方式来表示和区分文档的性质与格式，这使得浏览器能够知道应该以何种方式来处理相应类型的文档。为了让浏览器能够正确地处理远程位置上的 Wasm 模块，我们可以通过下面给出的一个自定义的基于 Node.js 编写的小型服务器程序来提供 HTTP 服务。通过这个小型的 HTTP 服务器，我们便可以将从远程地址下载的 Wasm 模块的响应内容类型标识为“application/wasm”。HTTP 服务器程序的代码如下（运行于 Node v8.9.1）。

```
server.js
// 引入使用到的外部模块文件
var http = require('http');
var url = require('url');
var fs = require('fs');
var path = require('path');
// 定义服务器端口号
const PORT = 8888;
// 定义对应文件的 MIME 类型映射数据
var mime = {
  "html": "text/html;charset=UTF-8",
  "wasm": "application/wasm"
};
// 启动一个 HTTP 服务器
http.createServer((request, response) => {
  var realPath = path.join(__dirname, `.${url.parse(request.url).pathname}`);
  // 如果对应远程位置的资源存在
  fs.access(realPath, fs.constants.R_OK, err => {
    // 如果资源不存在，则返回 404 状态
    if (err) {
      response.writeHead(404, {
        'Content-Type': 'text/plain'
      });
      response.end();
    } else {
      fs.readFile(realPath, "binary", (err, file) => {
        // 如果资源读取错误，则返回 500 状态
        if (err) {
```

```

    response.writeHead(500, {
      'Content-Type': 'text/plain'
    });
    response.end(err.code);
  } else {
    var ext = path.extname(realPath);
    ext = ext ? ext.slice(1) : 'unknown';
    // 根据请求文件的后缀来返回对应文件的 MIME 头
    var contentType = mime[ext] || "text/plain";
    response.writeHead(200, {
      'Content-Type': contentType
    });
    response.write(file, "binary");
    response.end();
  }
});
}
});
}).listen(PORT);
console.log("Server is running at port: " + PORT + ".")

```

我们直接在本地命令行中通过 `node` 命令来启动这个小型的 HTTP 服务器，然后通过 Web 浏览器来访问含有上述 JavaScript 脚本代码的 HTML 文件。经过 Wasm 模块的远程加载和实例化后，我们便可以在浏览器的控制台中看到如图 2-5 所示的打印结果。这里打印出了 Wasm 模块从开始加载到完成实例化过程消耗的总时间（以毫秒为单位），以及该 Wasm 模块经过实例化后 `WebAssembly.Instance` 对象的内容。

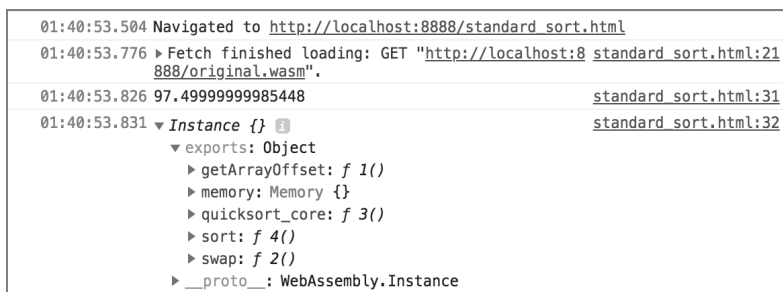


图2-5 上述Wasm应用在浏览器中的实际运行结果（使用`compileStreaming`方法改写）

WebAssembly.instantiateStreaming(source, importObject)

该方法的功能与 `WebAssembly.compileStreaming` 类似，可用于直接从系统底层的数据流源

来编译并实例化一个 Wasm 模块。在调用该方法时需要传入两个参数，其中第一个参数为用于加载 Wasm 模块数据的数据流源；第二个参数是 Wasm 模块实例化时需要导入的对象信息。我们可以通过下面给出的例子来了解 `WebAssembly.instantiateStreaming` 方法的实际用法。

```
let startTime = performance.now()
// 通过 WebAssembly.instantiateStreaming 方法直接从系统底层的数据流源编译并实例化一个 Wasm 模块
WebAssembly.instantiateStreaming(fetch('original.wasm'), {
  env: {
    print (offset, len) {
      let strBuffer = new Uint32Array(memory.buffer, offset, len);
      console.log(strBuffer);
    }
  }
}).then(resultObject => {
  console.log(performance.now() - startTime);
  console.log(resultObject)
});
```

在调用 `WebAssembly.instantiateStreaming` 方法时，仍然需要使用 HTTP 服务器来提供后端服务。应用运行完毕后，我们可以在浏览器控制台中看到如图 2-6 所示的运行结果。从图中可以看到，通过该方法来加载和初始化 Wasm 模块所花费的总体时间要少于之前的 `WebAssembly.compileStreaming` 方式。因此，这也是我们首先推荐使用的模块初始化方式。

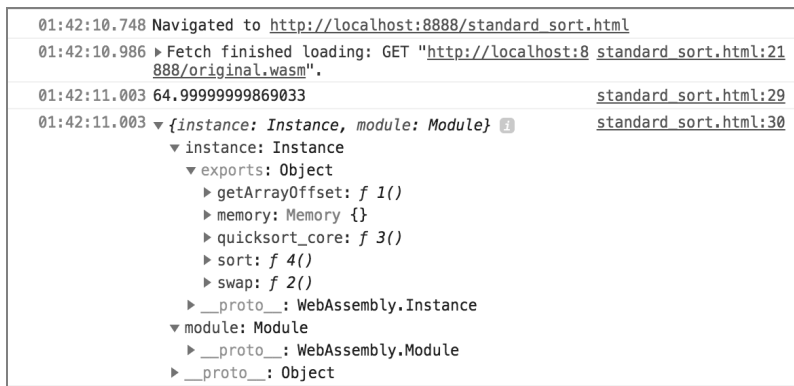


图2-6 上述Wasm应用在浏览器中的实际运行结果（使用instantiateStreaming方法改写）

至此，就介绍完了现有 MVP 标准中提供的用于编译和实例化 Wasm 模块的所有 JavaScript 标准库方法。我们将这些方法及各 WebAssembly 对象间的调用和转换关系整理在一起，如图 2-7 所示。

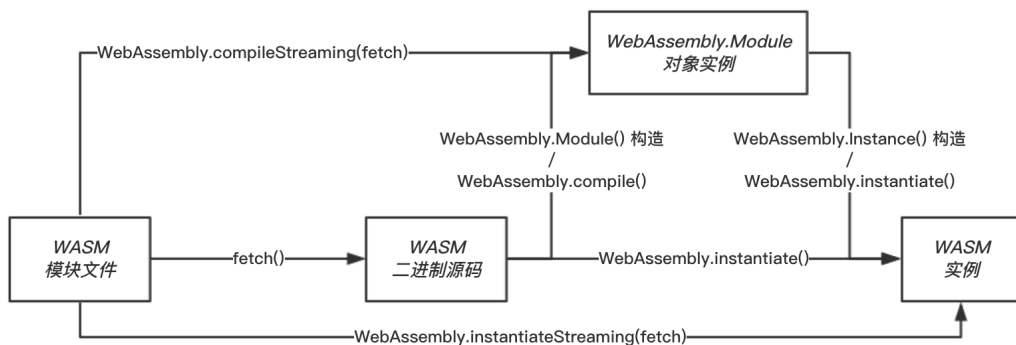


图2-7 Wasm标准上层JavaScript接口间的调用和转换关系

2.1.2 验证模块

WebAssembly.validate(bufferSource)

我们可以使用 `WebAssembly.validate` 接口来验证一个给定的 `TypedArray` 类型二进制数组或 `ArrayBuffer` 二进制缓冲区中的数据是否来自一个有效的 Wasm 模块。该接口的使用方法如下面的代码所示。

```
fetch('original.wasm').then(response => response.arrayBuffer()).then(bytes => {
  console.log(WebAssembly.validate(bytes));
});
```

如果 `WebAssembly.validate` 接口验证这些二进制数据确实来自一个标准有效的 Wasm 模块，即这些二进制数据可以被实例化为一个有状态的、可用的 `WebAssembly.Instance` 对象，则接口会返回 `true` 作为结果，否则返回 `false`。

2.1.3 遇到错误

在对 Wasm 模块进行验证、编译和实例化的过程中，难免会遇到一些异常或错误。比如在调用标准接口进行模块实例化时传递了错误的数据类型作为参数，这时浏览器便无法完成模块的实例化操作，同时会向用户抛出错误信息。在 `WebAssembly` 的 `Web` 标准接口中，规定了以下三种错误类型，浏览器会使用这三种错误类型来标识一个 Wasm 模块在其生命周期内发生的所有异常和错误。

WebAssembly.CompileError(message, filename, lineNumber)

该错误类型表示错误发生在 Wasm 模块的解码及验证阶段。

WebAssembly.LinkError(message, filename, lineNumber)

该错误类型表示错误发生在 Wasm 模块的实例化阶段。

WebAssembly.RuntimeError(message, filename, lineNumber)

该错误类型表示错误发生在 Wasm 模块的运行阶段。

这三种错误类型分别对应三种“错误对象”构造函数。在每一个错误对象中，都包含着该错误发生时产生的调用堆栈信息、代码位置、脚本文件名以及详细的错误信息等属性。这里我们以 `WebAssembly.CompileError` 为例来手动构造一个错误对象，并打印出该错误对象的属性信息。该构造函数支持传入三个参数，分别为用于描述该错误的详细信息、错误所在的脚本文件名以及错误所在的代码行号。详细代码如下。

```
compile_error.html
<!DOCTYPE html>
<html>
<head>
  <title>WebAssembly.CompileError 错误对象</title>
</head>
<body>
<script type="text/javascript">
try {
  // 手动抛出一个 WebAssembly.CompileError 错误对象，并立即捕获
  throw new WebAssembly.CompileError('WebAssembly.CompileError', 'compile_error.html',
10);
} catch (e) {
  // 打印出该 WebAssembly.CompileError 错误对象内部的所有属性信息
  console.log(e instanceof WebAssembly.CompileError);
  console.log(e.message);
  console.log(e.name);
  console.log(e.fileName);
  console.log(e.lineNumber);
  console.log(e.columnNumber);
  console.log(e.stack);
}
</script>
</body>
</html>
```

通过一个本地的 HTTP 静态服务器来提供 Web 服务。在浏览器中打开上面给出的 HTML

页面，从浏览器的 Console 控制台中可以看到我们手动捕获的 `WebAssembly.CompileError` 错误对象内部定义的所有属性信息，如图 2-8 所示。需要注意的是，当前浏览器还暂未完全支持 Wasm 错误对象在其标准中规定的所有属性。因此，图中诸如 `lineNumber` 以及 `fileName` 等属性的打印值均为 `undefined`。

01:32:44.250	true
01:32:44.251	WebAssembly.CompileError
01:32:44.251	CompileError
01:32:44.251	undefined
01:32:44.251	undefined
01:32:44.251	undefined
01:32:44.251	CompileError: WebAssembly.CompileError at <anonymous>:3:9
< 01:32:44.275	undefined
>	

图2-8 捕获到的WebAssembly.CompileError对象

2.1.4 内存分配

在通常情况下，我们会通过直接操纵共享线性内存的方式在 Wasm 模块对象实例和 JavaScript 环境之间传递复杂的数据结构，比如 C/C++ 数组或原生 JavaScript 对象中的元素。这块共享的线性内存存在 Wasm 模块实例化后便有了固定的起始地址和大小。随着 Wasm 应用的运行，共享内存可能会由于剩余资源不足而导致内存溢出的问题。这时我们可以使用 WebAssembly MVP 标准在 JavaScript 标准库中提供的相关接口，对这块共享内存的大小做出动态调整。

由于采用了内存分页机制，因此当通过 JavaScript 标准接口对这块共享内存进行资源分配操作时，需要以“页”为单位来计算实际分配的内存大小。在 WebAssembly 标准中规定，一个 Wasm 内存页所占用的实际内存大小为 64KB，即 65 536 字节。

在现代计算机组成结构中，内存分页机制是一种重要的内存管理机制。在计算机内部，CPU（中央处理器）是通过地址总线寻址的方式来直接访问物理内存的。比如一个基于 x86 架构的 CPU 其地址总线宽度为 32 位，即可寻址范围从位置 0 开始到 0xFFFFFFFF 结束。经过换算得知该 CPU 能够支持的最大物理内存大小为 4GB。在内存分页机制出现之前，我们经常会遇到这样的问题：一个应用程序需要使用 4GB 的内存来运行，但计算机实际可用的物理内存仅有 2GB，这导致开发者不得不重新编写该程序，以降低内存使用空间。而为了解决此类问题，在现代 CPU 中便引入了 MMU（Memory Management Unit，内存管理单元）。

MMU 的核心思想是使用虚拟内存地址来代替物理内存地址，即 CPU 在寻址时使用虚拟内

存地址（即逻辑地址），MMU 会负责将这个虚拟内存地址映射到真实的物理内存地址。MMU 的引入，解决了物理内存对应用程序的限制，MMU 会根据操作系统所支持的最大内存范围将应用程序使用的虚拟内存地址映射到物理内存地址。而内存分页机制便是在 MMU 的基础上提出的一种内存管理机制。内存分页机制会将虚拟内存地址和物理内存地址按照固定大小分别分割成“页”和“页帧”两种形式，并保证页与页帧的大小相同。页与页帧的出现使得操作系统可以进行非连续性的内存分配，因为应用程序所占用的内存实际物理介质上可能并不是连续存储的，以“页帧”为单位的内存段可能分散在整块物理线性内存的各个地方。另外，基于内存分页机制，当操作系统本身的可用物理内存容量不够时，CPU 还可以将一些不常用的物理内存页帧暂时转移到其他的线性存储设备上，比如硬盘。

MMU 在进行虚拟内存地址到物理内存地址映射时，需要一个名为“页表”的映射关系记录来找到对应的物理内存地址。在页表中，记录着所有虚拟内存页地址到物理内存页地址的映射关系。同时，页表本身也被存放在物理内存中。我们知道，CPU 通过地址总线访问物理内存的速度要慢于直接访问寄存器的速度。因此，为了进一步优化虚拟内存页地址的解析速度，在现代 CPU 中又引入了 TLB（Translation Lookaside Buffer，页表寄存器缓冲）寄存器来缓存一部分被经常访问的页表内容。我们可以通过图 2-9 来了解 CPU 从物理内存中查找数据的细节流程。

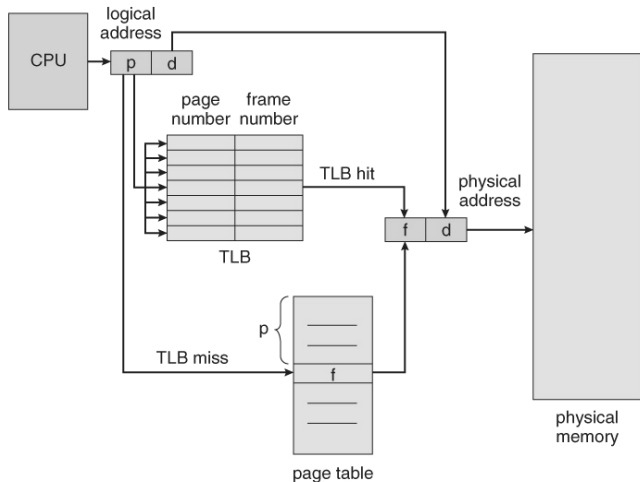


图2-9 CPU从物理内存中查找数据的细节流程

CPU 首先会从应用程序那里得到一个内存读取指令，在该指令中包含了一个对应于内存位置的虚拟内存地址，即逻辑地址。而这个虚拟内存地址本身又由两部分组成：该虚拟内存地址所在内存页的页编号（ p ）和地址在该内存页中的实际偏移量（ d ）。然后 CPU 会在 TLB 寄存器

中查找是否存在该内存页编号对应的页帧地址（f），即该页的实际物理内存地址。如果存在，则 CPU 会直接通过该页帧地址和该内存页的页内偏移量，来计算这个内存读取指令指向的最终实际物理内存地址。但如果 TLB 寄存器中没有该内存页的页帧地址，则 CPU 会直接从存放在物理内存中的页表来查找页帧地址。因此，从某种角度来看，如果能够适当提高内存的分页大小，则可以让 TLB 寄存器存储更多的地址映射关系，这会增加 TLB 的命中率，同时提高 CPU 执行内存读写指令时的性能。

WebAssembly.Memory(memoryDescriptor)

在 JavaScript 标准库中，我们使用 WebAssembly.Memory 对象来表示一个可以在 Wasm 模块对象实例和 JavaScript 环境中共享的线性内存段。WebAssembly.Memory 对象内部是一个 ArrayBuffer 形式的二进制缓冲区，在这个缓冲区中包含了以二进制形式存在的内存比特流数据。我们可以从 JavaScript 环境或 Wasm 模块对象内部直接操作这个共享缓冲区中的数据。可以通过如下方式来创建一个 WebAssembly.Memory 对象。

```
// 声明一个 WebAssembly.Memory 对象，表示一段可以在 Wasm 模块对象实例与 JavaScript 环境中共享的内存
let memory = new WebAssembly.Memory({
  // 设置该内存段的初始大小和最大可用大小
  initial: 10,
  maximum: 100
});
```

WebAssembly.Memory 在作为构造函数使用时，需要接收一个名为“memoryDescriptor”的参数。该参数以原生 JavaScript 对象的形式封装了一些用于控制 WebAssembly.Memory 对象自身特性的属性。在该对象中，我们可以通过几个特定的属性来指定该 WebAssembly.Memory 对象对应实际物理内存段的初始大小和最大可用大小。比如在上述代码中，通过 initial 属性指定了该 WebAssembly.Memory 对象内部的 ArrayBuffer 结构在初始化时的大小为 10 个 WebAssembly 内存页，即 640KB 大小；通过 maximum 属性指定了该 ArrayBuffer 结构的最大可用内存大小，这里将其设置为 100 个 WebAssembly 内存页，即 6400KB 大小。在实际的 Wasm 应用中，我们可以通过两种方式来获得一个 WebAssembly.Memory 对象。

第一种方式是直接使用 WebAssembly.Memory 构造函数，从 JavaScript 环境手动构建一个 WebAssembly.Memory 对象，如上面代码所示。通过这种方式构建的 WebAssembly.Memory 对象可以在 Wasm 模块进行实例化时作为导入参数传递给该模块，这样模块便可以直接使用这个 WebAssembly.Memory 对象中的线性内存来分配资源了。而对于那些没有在实例化时传入外部自定义的 WebAssembly.Memory 对象的 Wasm 模块，模块本身会在实例化时自动使用系统默认

指定的共享线性内存段（通常为一个 WebAssembly 内存页，即 64KB 大小）来分配资源。

第二种方式是从 Wasm 模块对应 WebAssembly.Instance 对象的导出对象中来获得一个 WebAssembly.Memory 共享内存对象。在 WebAssembly.Instance 对象的 exports 属性中含有从 Wasm 模块导出到 JavaScript 环境中的所有对象信息，而在这些导出对象中就包含了 Wasm 模块当前使用的共享内存对象，对应到 JavaScript 环境中即为一个 WebAssembly.Memory 对象。

无论是让 Wasm 模块在实例化时使用系统为其默认分配的共享内存段，还是使用我们在 JavaScript 环境中手动构建的具有固定大小的共享内存段，这两者在使用的最终效果上没有任何区别。WebAssembly.Memory 对象的主要作用就是用于在 JavaScript 环境和 Wasm 模块实例之间传递数据。在一些类似 Redis 的内存型数据库中，为了能够实现数据的快速存取，通常会吧写入数据库的数据直接缓存在内存中。Redis 会在其内部维护一个全局变量来标识当前内存中所有已写入数据占用的总大小，该变量的值会在每次数据写入时增大，数据移除时减小。通过该变量，Redis 可以在本地内存耗尽前向用户及时发出警告，从而防止出现数据丢失的问题。

此时我们通过 WebAssembly 技术将 Redis 数据库移植到 Web 平台上，那么当 Redis 对应的 Wasm 模块实例所使用的共享内存超过其模块初始内存大小时，我们便可以通过 WebAssembly.Memory 对象上的 grow 方法来对这块共享内存的大小进行动态调整。但需要注意的是，调整后的内存大小不能超过由 maximum 属性设置的最大可用内存大小。下面的代码给出了 grow 方法的使用方式。

```
// 手动构建一个 WebAssembly.Memory 对象。该对象代表一个 Wasm 模块使用的共享内存段
let memory = new WebAssembly.Memory({
  // 设置该 WebAssembly.Memory 对象对应内存段的初始大小为 1 个内存页，最大可用大小为 10 个内存页
  initial: 1,
  maximum: 10
});
// 常量用于存储一个 WebAssembly 内存页的大小，即 64KB
const bytesPerPage = 64 * 1024;
// 打印共享内存段当前使用的内存页数
console.log(memory.buffer.byteLength / bytesPerPage);
// 通过标准接口将共享内存段的当前大小增加 10 个内存页
console.log(memory.grow(5));
console.log(memory.buffer.byteLength / bytesPerPage);
```

我们将上面代码直接放到浏览器的控制台中运行，可以看到如图 2-10 所示的运行结果。

01:32:10.768 1
01:32:10.768 1
01:32:10.768 6
< 01:32:10.793 undefined
>

图2-10 使用grow方法增加共享内存段的可用大小

`WebAssembly.Memory.prototype.grow` 是 `WebAssembly.Memory` 对象实例中的方法。通过该方法，我们可以在 JavaScript 环境下动态调整 Wasm 模块实例使用的共享内存段大小。在上面代码中，我们构建了一个初始大小为 1 个 WebAssembly 内存页的共享内存段，同时还指定了该共享内存段的最大可用大小为 10 个 WebAssembly 内存页。`grow` 方法仅接收一个参数，即对应的共享内存段需要增加的内存页数。这里我们通过调用 `grow(5)`，将之前构建的 `WebAssembly.Memory` 对象实例当前可用的共享内存段增加了 5 个 WebAssembly 内存页大小。`grow` 方法执行完毕后会返回共享内存段在增加内存页前的原始大小。

2.1.5 表

表（Table）是 WebAssembly 技术中一个复杂却十分重要的概念。与 `WebAssembly.Memory` 对象对应的共享内存段一样，从组成形式上看，表本身也代表一个具有固定大小且连续的线性内存段。不同的是，存放在表中的数据需要符合特定的 WebAssembly 数据类型，并且表结构对应的线性内存段无法以二进制数据的形式被直接读取。在现阶段的 MVP 标准中规定，存放在表中的数据必须是且只能是 `anyfunc` 类型，即带有不同函数签名的函数类型。在 WebAssembly 的整个技术体系和规范中，对表的定位主要是用于存储一些无法被前端用户直接读取和修改的符合“引擎可信”特征的数据。比如在 C/C++ 等编译型语言中，函数指针一般是指某个函数的函数体在当前程序虚拟地址空间中的偏移地址，通过函数指针可以找到该函数的函数体并进行函数调用操作。而所谓的“引擎可信”特征，是指对于 C/C++ 的运行引擎（这里指 C/C++ 应用的运行时环境）来说，每一个函数指针都一定会正确地对应着一个完整的函数体。

当我们将 C/C++ 应用通过 WebAssembly 技术转译到 Web 平台上运行时，出于安全性及稳定性等因素的考虑，Wasm 模块并不允许用户直接通过内存来修改函数体的函数指针。但是为了支持多个 Wasm 模块之间的动态链接特性，Wasm 标准允许将函数体的函数指针存放到表结构中。表会对存入其内部的数据进行类型检查，以判断这些数据是否是引擎可信的，即是否被人多地修改过。存入表结构中的每一项数据都会有一个整型的数字索引与之对应，通过该数字索引，我们可以在 JavaScript 环境中使用特定的表操作接口来获得对应索引位置处的表项数据。如果该表项数据是一个符合 `anyfunc` 类型的函数指针，那么我们便可以直接通过该函数指针来

进行函数调用。

在现阶段的 MVP 标准中，表结构具有以下几个特性。

- 只能间接地通过编辑 WebAssembly 可读文本（WAT）或二进制模块代码的方式来让 Wasm 模块使用表（内部生成或从外部 JavaScript 环境导入的），以及通过表结构进行诸如模块动态链接等高级特性。
- 在当前的表结构中仅允许存放类型为 `anyfunc` 的数据项，该类型的数据项代表一个指向特定函数的函数指针。
- 无法从 Wasm 模块内部直接动态修改一个表的结构、表项内容及表的大小。但是我们可以在 Wasm 模块实例化前后通过 JavaScript 标准接口来执行这些操作。

WAT 是 Wasm 二进制模块的可读文本代码格式。WAT 通过一套特定的语法格式和关键字以可读文本代码的形式描述了 Wasm 模块的执行逻辑和内容。如果将 Wasm 模块的二进制代码比作机器语言的话，那么 WAT 就相当于可以编译成机器语言的汇编语言。下面我们通过一个简单的例子，来了解如何通过 WebAssembly 可读文本（WAT）来生成一个 Wasm 模块。在这段 WAT 代码中，我们设置了模块默认表结构的初始大小为 2，并且在表结构索引值为 0 的位置处填充了一个名为“add”的函数的函数指针。我们将如下的 WAT 代码保存在一个以“.wat”为后缀的文本文件中，再通过 WABT（WebAssembly Binary Toolkit）工具链对这段代码进行编译。

```
wat2wasm_table.wat
```

```
(module
  (table $0 2 anyfunc)
  (memory $0 1)
  (export "memory" (memory $0))
  (export "add" (func $add))
  (export "minus" (func $minus))
  (export "table" (table $0))
  (func $add (; 0 ;) (param $0 i32) (param $1 i32) (result i32)
    (i32.add
      (get_local $1)
      (get_local $0)
    )
  )
  (func $minus (; 1 ;) (param $0 i32) (param $1 i32) (result i32)
    (i32.sub
```

```

    (get_local $0)
    (get_local $1)
  )
)
(elem (i32.const 0) $add)
)

```

WABT 工具链是专门用于对 WebAssembly 二进制模块本身进行分析和处理操作的工具集合。在本地开发环境中部署好 WABT 工具链后，我们便可以通过如下命令将上述 WAT 代码编译成一个标准的 Wasm 二进制模块。

```
wast2wasm wat2wasm_table.wat -o wat2wasm_table.wasm
```

执行上面命令，WABT 会在当前文件夹下生成一个名为“wat2wasm_table”的 Wasm 模块文件。接下来，我们通过 JavaScript 脚本代码在 Web 浏览器中整合、加载并实例化这个 Wasm 模块。这部分代码如下。

```
wat2wasm_table.html
```

```

<!DOCTYPE html>
<html>
<head>
  <title></title>
</head>
<body>
  <script>
    // 加载并实例化 Wasm 模块
    fetch('wat2wasm_table.wasm').then(response =>
      response.arrayBuffer()
    ).then(bytes =>
      WebAssembly.instantiate(bytes)
    ).then(results => {
      // 从 WebAssembly.Instance 对象中获得模块的导出对象
      let exportsObject = results.instance.exports;
      // 从 exports 属性中获得一个 WebAssembly.Table 对象
      let table = exportsObject.table;
      // 打印当前表结构的大小
      console.log("Table 原始长度: ", table.length);
      // 将表可以容纳的表项数量加 1
      table.grow(1);
      console.log("Table 增长后的长度: ", table.length);
    });
  </script>

```

```

// 从表的索引位置 0 处获取 add 函数的函数指针
let tableAddFunctionHandle = table.get(0);
// 通过函数指针调用该函数
console.log(tableAddFunctionHandle(1, 2));

// 在表的索引位置 1 填充新的表项，内容为函数 minus 的函数指针
table.set(1, exportsObject.minus);
// 调用导入到表中的 minus 函数
console.log(table.get(1)(1, 2));
});
</script>
</body>
</html>

```

在上述 JavaScript 脚本代码中，首先对 Wasm 模块进行了加载和实例化。然后在对应实例化模块的 `WebAssembly.Instance` 对象的 `exports` 属性中，我们获得了从模块导出的 `WebAssembly.Table` 对象。

WebAssembly.Table(tableDescriptor)

存在于 JavaScript 环境中的每一个 `WebAssembly.Table` 对象都代表了一个 Wasm 模块中的表结构实体。与 `WebAssembly.Memory` 对象一样，获得 `WebAssembly.Table` 对象也有两种主要方式。第一种方式，可以直接从 Wasm 模块实例中导出一个 `WebAssembly.Table` 对象；第二种方式，可以通过 `WebAssembly.Table` 构造函数在 JavaScript 环境下手动构造一个 `WebAssembly.Table` 对象。同样的，通过手动构造生成的 `WebAssembly.Table` 对象可以在 Wasm 模块进行实例化操作时作为参数传递给模块使用（模块是否会真正使用依赖模块本身）。`WebAssembly.Table` 构造函数接收一个原生 JavaScript 对象作为其参数，在该对象中包含了用于描述新生成表对象特征的属性。我们可以用如下代码来构建一个 `WebAssembly.Table` 对象。

```

var table = new WebAssembly.Table({
  // 指定该 WebAssembly.Table 对象的表结构可以存储的表项数量及表项类型
  initial: 2,
  element: "anyfunc"
});
console.log(table.length); // "2"
console.log(table.get(0)); // "null"

```

在现有的 MVP 标准中，`tableDescriptor` 参数的描述属性有两个：`initial` 属性用于设置该 `WebAssembly.Table` 对象初始时可存储的表项数量；`element` 属性用来指定该表结构可以存储的具体表项类型。`WebAssembly.Table` 对象还包含一些用于操作和处理表项及表结构的属性和方

法，对这些属性和方法的说明如下。

Table.prototype.length

该属性可以返回 `WebAssembly.Table` 对象对应表的当前长度，即在当前表结构中存储的表项元素数量。

Table.prototype.get(index)

该方法可以从 `WebAssembly.Table` 对象对应表结构的指定索引位置处获取一个表项内容。在当前 MVP 标准下，通过该方法获取到的表项元素均为 `anyfunc` 类型，即函数指针。

Table.prototype.grow(number)

该方法用于增加 `WebAssembly.Table` 对象对应表结构可存储表项的数量。该方法执行后，会返回表容量增加前的可存储表项数量。

Table.prototype.set(index,value)

该方法用于设置 `WebAssembly.Table` 对象对应表结构在指定索引位置处的表项内容。在当前 MVP 标准下，该方法只能设置类型为 `anyfunc` 的表项内容，即函数签名不同的函数指针。

在前面的例子中，我们直接使用了 `Wasm` 模块在初始化时系统默认为其生成的表。下面的例子我们将通过 `WebAssembly.Table` 构造函数从 `JavaScript` 环境手动构建一个表对象，并在模块实例化时将其作为参数传递给模块使用。首先还是通过改写 `Wasm` 模块对应的 `WAT` 代码，来让模块能够使用一个从外部 `JavaScript` 环境中传入的 `WebAssembly.Table` 对象作为其自身的表结构。代码如下。

```
wat2wasm_table_imported.wat
(module
  (import "env" "table" (table $0 1 anyfunc))
  (memory $0 1)
  (export "memory" (memory $0))
  (export "add" (func $add))
  (export "minus" (func $minus))
  (export "table" (table $0))
  (func $add (; 0 ;) (param $0 i32) (param $1 i32) (result i32)
    i32.add
    (get_local $1)
    (get_local $0))
```



```

)
)
(func $minus (; 1 ;) (param $0 i32) (param $1 i32) (result i32)
  (i32.sub
    (get_local $0)
    (get_local $1)
  )
)
(elem (i32.const 0) $add)
)

```

同样的，我们还是使用 WABT 工具链将这段 WAT 代码编译成一个标准的 Wasm 二进制模块。模块编译完成后，我们会在 JavaScript 脚本中将这个手动构建的表对象作为参数传递给实例化中的模块进行使用。JavaScript 部分的代码如下。注意：这里需要将传入的 `WebAssembly.Table` 对象放置在名为“env”的模块导入对象内。

wat2wasm_table_imported.html

```

<!DOCTYPE html>
<html>
<head>
  <title></title>
</head>
<body>
  <script>
    // 手动构建一个 WebAssembly.Table 对象
    let tbl = new WebAssembly.Table({
      initial: 10,
      element: "anyfunc"
    });

    // 加载并实例化一个 Wasm 模块
    fetch('wat2wasm_table_imported.wasm').then(response =>
      response.arrayBuffer()
    ).then(bytes =>
      WebAssembly.instantiate(bytes, {
        // 将 WebAssembly.Table 对象作为参数传递给实例化中的 Wasm 模块使用
        "env": {
          "table": tbl
        }
      })
    )
  </script>

```

```

).then(results => {
  let table = results.instance.exports.table;
  // 打印该 Wasm 模块使用的表结构的大小
  console.log("Table 原始长度: ", table.length); // "10"
  // 通过表在其索引位置 0 处的函数指针来调用对应函数
  console.log(table.get(0)(1, 2)); // "3"
});
</script>
</body>
</html>

```

总的来看，在现阶段的 MVP 标准下，Wasm 模块中的“表”特性所能够覆盖的应用场景还很少。并且由于无法直接在 C/C++ 代码中操作和设置表的相关属性，只能通过修改模块 WAT 代码的方式，使得表本身的应用成本又进一步增加。诸如“Wasm 模块间动态链接”，以及“Wasm 模块重载函数调用容器”等表特性的应用，我们会在后续章节中进行介绍。

至此，就介绍完了在 WebAssembly MVP 标准中制定并且已经在浏览器端实现的所有标准上层 JavaScript 接口。接下来，让我们走进 Wasm 模块的底层世界，去看一看 WebAssembly 应用的内部运行机制和原理。

2.2 深入设计模型——堆栈机

WebAssembly 之所以会有如此高的加载和运行效率，离不开 Web 浏览器在其底层对 Wasm 代码独特的解析和执行方式。如图 2-11 所示，WebAssembly 直接使用浏览器内部的 JavaScript 虚拟机（VM）作为其执行的宿主环境。在该虚拟机的内部执行环境中，JavaScript 和 Wasm 分别对应着不同的底层编译器，而两者会在某种程度上共享一部分通用的编译器后端。

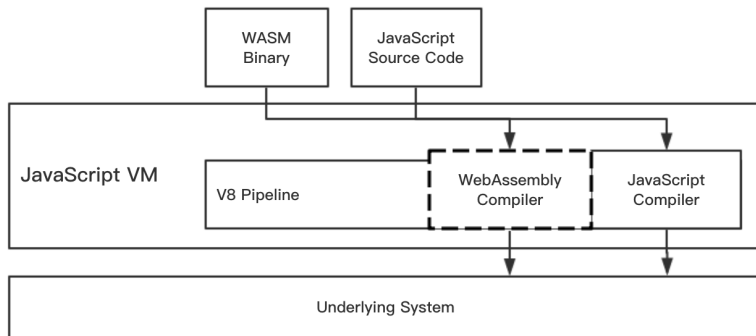


图2-11 浏览器底层JavaScript VM与WebAssembly的交互流程图

JavaScript 虚拟机是一种进程类虚拟机，即该虚拟机环境会在 Web 浏览器启动时随之自动生成，而当浏览器关闭时也会随之销毁。进程类虚拟机的主要作用是为某种编程语言提供一个平台无关的运行时环境。虚拟机会将底层硬件和操作系统本身的细节进行高度抽象，抽象出来的通用特性会被直接整合到虚拟机的运行时环境中供上层的编程语言使用。通过虚拟机提供的抽象的运行时环境，使用该编程语言编写的应用程序便可以在基于各类处理器架构的操作系统环境中无缝地移植和运行。总的来讲，进程类虚拟机为编程语言屏蔽了底层系统和处理器架构上的细节差异。对上层来说，虚拟机为编程语言提供抽象的运行时环境；对下层来说，虚拟机会将特定编程语言的代码解释编译成具体平台上的机器指令并加以执行。这种特性让基于该编程语言编写的应用可以真正做到“一次编写，到处运行”。

与 JVM（Java Virtual Machine，Java 虚拟机）一样，JavaScript 虚拟机为 JavaScript 语言提供了一个抽象且平台无关的运行时环境。不同的是，当 JVM 在操作系统中运行时，它有着自己的独立进程、生命周期，以及代码和数据空间等资源；而 JavaScript 虚拟机则是作为 Web 浏览器底层基础设施的一部分来实现的，因此它并没有自己的独立进程和环境资源。

浏览器中用于编译 Wasm 二进制模块的编译器不同于通常的 C/C++ 编译器。比如在使用 V8 引擎内置的 TurboFan 编译器编译 Wasm 模块时会大致分为 7 个步骤，其中包括对模块内函数体的解码过程、SSA 图的构建过程、代码优化过程、资源分配与调度过程、指令选择过程、寄存器分配过程以及最后的机器代码生成过程。

在整个 JavaScript 虚拟机的组件体系中，编译器会根据不同的语法和语义规则来编译和优化 WebAssembly 和 JavaScript 两种目标代码。我们可以将 Wasm 代码在编译器中的处理和执行过程抽象为一种新的虚拟机类型，称之为“Wasm 抽象虚拟机”。Wasm 抽象虚拟机是对浏览器按照 MVP 标准编译和执行 Wasm 代码过程的一种抽象。单纯地从 Wasm 二进制代码在编译器内部的解析和处理流程来看，这种对代码的处理和执行方式与我们经常见到的一种虚拟机类型十分相似，它就是“堆栈式虚拟机”。WebAssembly 的设计和对编译器的实现在某种程度上借鉴了堆栈式虚拟机的结构化控制流和栈容器等特性。

2.2.1 堆栈式虚拟机

在下文中我们会将堆栈式虚拟机简称为“堆栈机”。顾名思义，堆栈机是指虚拟机在执行指令时使用“栈”这种容器结构来存储和交换数据。栈是一种“后进先出”的数据结构，即最后被放入栈容器中的数据可以被最先取出。

堆栈机中的大部分指令在执行时都会从栈容器中取出操作数，然后根据指令的功能，堆栈机会对这些操作数进行相应的数学或逻辑运算。当运算过程结束后，所得到的结果会被重新压入栈容器中。比如我们可以通过图 2-12 中给出的三条指令来演示堆栈机的基本工作流程。

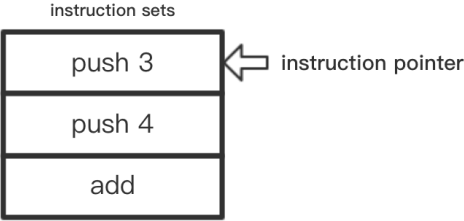


图2-12 堆栈机的基本工作流程

一般来说，在堆栈机的具体实现过程中，源代码中的每一条语句都可能会对应底层的多条虚拟机指令（OpCode，Operation Code 的缩写）。我们会将源代码中每条语句对应的一簇虚拟机指令统称为一个指令集合，而多个指令集合便组成了一个全局的指令队列。在指令队列中，虚拟机会在其内部维护一个全局指令指针来指向下一条将要执行的指令所在位置。在指令的执行阶段，虚拟机会从指令队列的队首开始依次逐条执行每一条指令。一条指令执行完毕后，指令指针的内容也会随之改变，即指向指令队列中下一条将要执行指令的所在位置。

图 2-12 所示的三条指令正好可以用来模拟表达式 “ $A=3+4$ ” 在堆栈机中的具体执行过程。首先，堆栈机会从该指令集合的第一条指令开始依次向后执行。当第一条指令执行完毕后，堆栈机会将数字 “3” 压入用于存放数据的栈容器中。此时指令队列和栈容器的状态如图 2-13 所示。

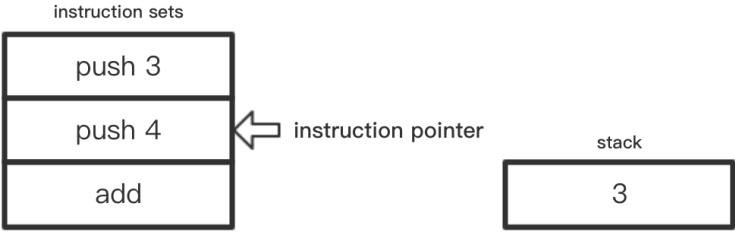


图2-13 执行 “push 3” 指令后指令队列和栈容器的状态

可以看到，此时在栈容器中只有一个值为 “3” 的操作数。这时指令指针向后移动，指向了指令集合中的第二条指令，堆栈机继续执行该条指令，把操作数 “4” 压入栈容器中。该条指令执行完毕后，指令队列和栈容器的状态如图 2-14 所示。

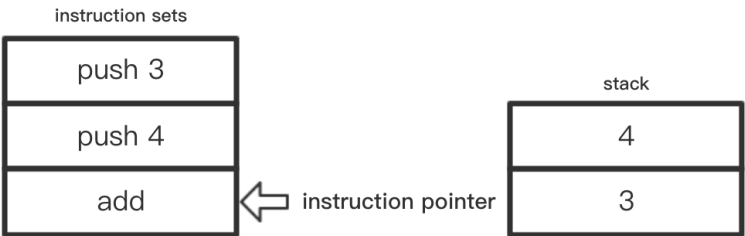


图2-14 执行“push 4”指令后指令队列和栈容器的状态

可以看到，此时栈容器中操作数之间的顺序关系。在堆栈机中，无论是向栈容器中存放数据，还是从栈容器中取出数据，都只能从栈容器的顶部来进行操作。也正是因为栈这种独有的结构，才使其具有 LIFO（Last In First Out，后进先出）的特性。接下来，堆栈机继续执行指令集合中的最后一条指令。通过这条 `add` 指令，栈容器中位于栈顶的两个操作数会进行加法操作。该条指令执行完毕后，指令队列和栈容器的状态如图 2-15 所示。



图2-15 执行add指令后指令队列和栈容器的状态

堆栈机在执行 `add` 指令时会首先判断所需要的操作数个数，然后依次从栈容器中取出对应数量的操作数。这里 `add` 指令需要两个操作数，因此堆栈机会将栈容器顶部的前两个元素，也就是之前压入栈容器中的两个操作数依次取出，并进行计算和处理。在计算完毕后，所得到的计算结果会被再次压入栈容器中。不仅如此，当堆栈机发现当前指令集合中的指令已经全部执行完毕时，堆栈机会将位于栈容器顶部的第一个元素取出作为本次指令集合的返回值。在这里，栈容器中最后存放的数字“7”便被作为表达式“`3+4`”的返回值赋值给变量“`A`”。

以上便是 Wasm 在标准层面使用堆栈式虚拟机模型基于栈结构执行指令的大致过程。但实际上，如果按照指令在虚拟机中的执行方式来区分的话，还有另外两种虚拟机模型也会被经常使用到，即“累加器型虚拟机”和“寄存器型虚拟机”。下面我们将简单介绍这两种虚拟机各自的指令执行方式，并对比说明三种虚拟机模型各自的特点。

累加器型虚拟机

累加器型虚拟机是一种比较古老的虚拟机模型。在执行指令时，它会使用 CPU 上一个特定的累加器寄存器来作为操作数的暂存容器。因此，虚拟机的每一条指令最多只能拥有一个操作数，即只能对累加器中存放的数据进行一次简单的读写或二元运算操作。也正是由于这个原因，基于累加器实现的虚拟机在执行复杂运算指令时可能会消耗大量的本地线性内存，这些线性内存通常会作为暂存容器来存储指令执行时产生的中间数据。但相对而言，由于累加器型虚拟机具有简单的指令执行过程和数据交换方式，最大限度地简化了虚拟机内部可能存在的状态类型，同时也相应地缩短了指令的长度。如果通过累加器型虚拟机来执行语句“A=3+4”，其执行流程大致如图 2-16 所示。

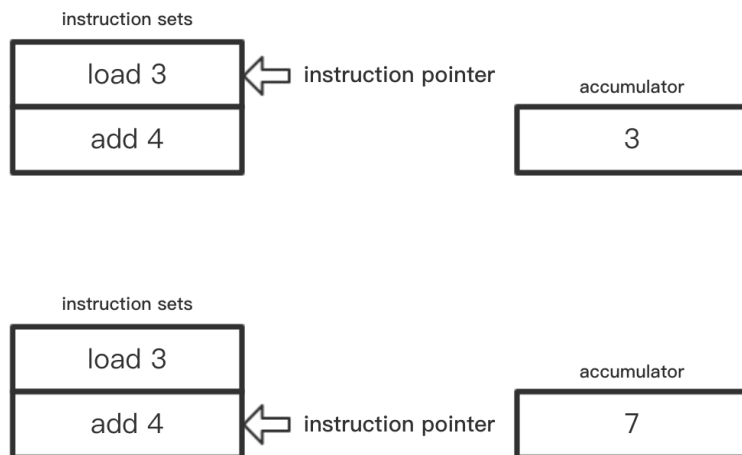


图2-16 累加器型虚拟机的指令执行流程

可以看到，指令集合中的所有指令都只有一个操作数。执行第一条指令“load 3”，虚拟机会将数字“3”存放到其内部的累加器寄存器中。执行第二条指令“add 4”，虚拟机会将当前累加器寄存器中的数字“3”取出并累加数字“4”。当 add 指令执行完毕后，所得到的执行结果会被重新写入当前的累加器寄存器中。当一条语句对应的指令集合全部执行完毕后，虚拟机会将当前累加器寄存器中的值返回给该语句作为执行的最终结果。这里虚拟机将最后存放在累加器寄存器中的数字“7”作为结果返回并赋值给变量“A”。

寄存器型虚拟机

寄存器型虚拟机使用特定的 CPU 寄存器组来作为指令执行过程中数据的暂存容器。在寄存器型虚拟机中，需要为每一条指令都指定其操作数所在的寄存器地址，因此与堆栈机和累加器型虚拟机相比，寄存器型虚拟机的平均指令更长，所生成的虚拟机指令集合代码更多。但另一

方面，由于寄存器型虚拟机拥有多个可以用来暂存数据的容器单元，因此对于某些复杂的语句，虚拟机可以仅通过一条指令来执行，而不需要数据交换过程。除此之外，寄存器型虚拟机还可以对某些运算流程进行优化，而优化策略均得益于其拥有的众多数据暂存容器。下面使用寄存器型虚拟机来模拟语句“A=3+4”的执行流程，如图 2-17 所示。

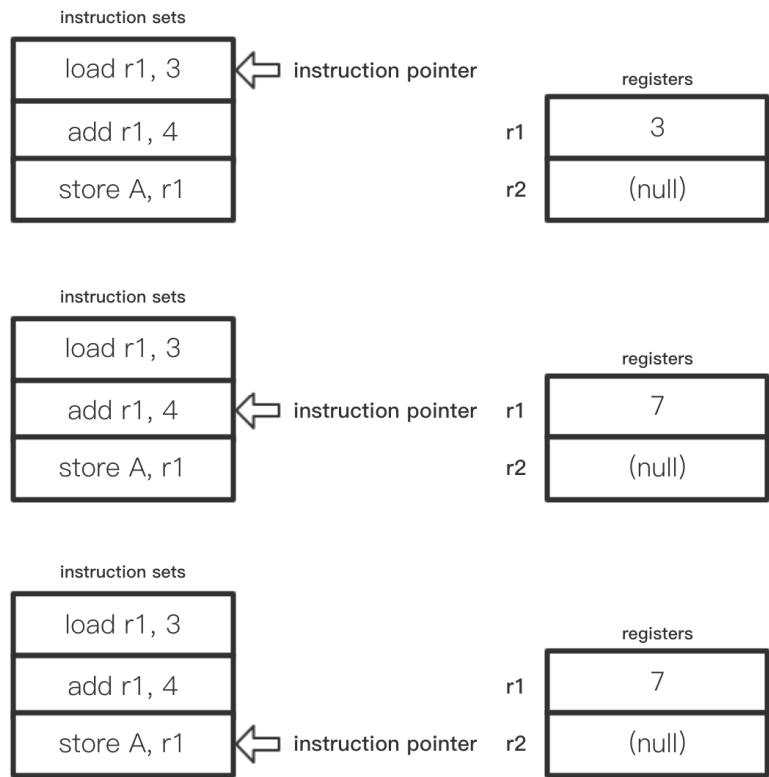


图2-17 寄存器型虚拟机的指令执行流程

可以看到，这里假设虚拟机内部拥有两个用于暂存操作数的寄存器，r1 和 r2 分别代表两个寄存器的地址，也可以将其理解为寄存器别名。执行第一条指令“load r1, 3”，虚拟机将数字“3”存储到 r1 寄存器中。执行第二条指令“add r1, 4”，将数字“4”累加到 r1 寄存器中，此时 r1 寄存器中存放的数字值由原来的“3”变成了“7”。执行最后一条指令，虚拟机会直接将 r1 寄存器中存放的数字值直接赋值给变量“A”。在整个指令集合的执行过程中，由于 r2 寄存器并没有被使用到，因此其内部一直都保持在空值的状态。

总的来看，三种不同的虚拟机模型都有其各自的实现方式和相应的优缺点。堆栈机使用栈结构来作为暂存数据的容器，这使得我们无法对栈容器中的数据进行任意读取，我们需要

遵循 LIFO 的数据操作原则来对数据进行处理。这导致无法从源代码直接生成最高效的虚拟机代码，因为对于某些较为复杂的语句来说，可能会涉及栈数据频繁交换的过程。但另一方面，基于栈结构实现的虚拟机模型最简单，所生成的虚拟机代码密度适中。

累加器型虚拟机内部只有一个累加器寄存器作为数据的暂存容器，因此在指令的执行过程中会大量占用本地线性内存。同时由于内存的数据交换速度相对寄存器较慢，这也使得指令的执行效率大大降低。但是简单的数据和指令执行流程使得累加器型虚拟机内部可以保持最少的状态和最简短的指令。

寄存器型虚拟机将操作数存储在多个不同的寄存器单元上，这使得每条指令在执行时都需要指定操作数所在的寄存器地址，而这无疑增加了指令的长度，同时也使虚拟机的实现变得复杂。但是存在多个寄存器单元为寄存器型虚拟机提供了更大的优化空间，因此可以生成执行效率更高的虚拟机代码。

而 WebAssembly 之所以会选择使用堆栈机模型来设计和执行其内部虚拟指令，除堆栈机本身具有实现简单、快速等特性外，它还会使 Wasm 模块的代码验证过程变得更简单和高效。

浏览器在加载和编译一个 Wasm 模块前，会首先检查模块中的代码是否符合 WebAssembly 标准，以及相应的安全特性。比如模块是否访问了规定范围外的内存、模块中各个函数的返回值类型是否正确，以及块语句中变量的作用域是否正确等。而基于堆栈机模型，我们便可以十分简单和快速地进行这些检查。

比如函数执行完毕后，编译器可以直接通过检查栈容器中位于其栈顶元素的实际数据类型，来判断模块中该函数标识的返回值类型是否正确。还有，块语句（循环/条件语句）中的代码并不会改变栈容器中的数据，因此，只要判断在块语句执行前后栈容器中的数据签名是否一致，即可判断出模块中的块语句是否表达正确。除此之外，基于堆栈机的结构化控制流，使得 Wasm 模块内部的二进制代码可以被高效地解码成编译器内部的 SSA（Single Static Assignment，静态单赋值）代码，这在某种程度上也简化了浏览器对 WebAssembly 二进制代码的编译和分析过程。

SSA 是一种用在编译器后端的数据流分析技术。通过该技术，可以让编译器的数据流分析和优化算法更简单。SSA 会使用唯一的名称（比如“变量名+变量被赋值的次数”）来代替程序控制流中所有赋值语句中的变量，这样可以保证每个变量都具有唯一的定义。即通过 SSA 技术，编译器可以获得精确的变量“使用-定义”关系。这种明确的关系有利于编译器进行更精确、彻底和高效的代码优化，甚至是底层寄存器的分配优化等过程。

2.2.2 逆波兰表达式

本节我们继续深入了解堆栈机内部的具体工作细节。大多数基于栈容器实现的虚拟机，都会选择使用两种常见的堆栈机形式，即“RPN 计数器”和“PN 计数器”。

通常，我们在书写一个数学表达式时会使用“中缀表示法”来进行描述，比如表达式“ $1 + 1$ ”表示对两个数字“1”进行相加操作。中缀表达式通常用来描述那些将运算符放置在操作数中间的表达式类型，这种表达式的书写方式也经常被各种编程语言所使用。但中缀表示法的缺点之一是对于某些复杂的表达式（比如“ $1 + 2 * 3$ ”），其运算规则并不明确，虚拟机无法知道应该优先进行哪一个运算符的运算。此时虚拟机需要结合运算符的运算优先级规则，才能够正确地解释表达式。

而使用基于 RPN 和 PN 形式的堆栈机模型便可以解决这个问题。RPN 是 Reverse Polish Notation（逆波兰表示法）的英文缩写。RPN 表达式也可以被称为“后缀表达式”。与中缀表达式一样，RPN 的语法规则：表达式中的每一个运算符都应该被放置到其所有操作数的后面。如果将上述基于中缀表示法的表达式“ $1 + 1$ ”改写成基于 RPN 形式的表达式，那么便变成了“ $1\ 1\ +$ ”。同理，中缀表达式“ $1 + 2 * 3$ ”的 RPN 形式为“ $2\ 3\ * \ 1\ +$ ”。对于该表达式，我们需要按照从左至右的顺序依次计算出每个运算符和相应操作数的运算结果，然后再将表达式依次向后展开。图 2-18 展示了 RPN 表达式“ $2\ 3\ * \ 1\ +$ ”的具体运算流程。

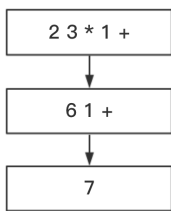


图2-18 RPN表达式“ $2\ 3\ * \ 1\ +$ ”的具体运算流程

可以看到，基于 RPN 形式的表达式可以很清楚地让虚拟机了解到整个表达式的具体运算流程，而不再需要进行额外的运算符优先级判断。不仅如此，虚拟机还可以十分简单地通过一个栈容器来实现 RPN 表达式对应虚拟指令的解析和执行。比如对于 RPN 表达式“ $2\ 3\ * \ 1\ +$ ”，虚拟机可以从左至右按顺序解析执行每一个子表达式并使用栈容器保存中间结果值。该表达式的执行流程和栈容器的状态变化如图 2-19 所示。

在图 2-19 中，我们将该 RPN 表达式中每一个元素对应的底层虚拟指令列了出来。可以看到，虚拟机会直接按照从左到右的顺序来解析该 RPN 表达式，而表达式中的每一个元素都会直

接对应到虚拟机内部的一个底层虚拟指令，也就是我们经常提到的 **OpCode**。通过执行这些虚拟指令，虚拟机可以直接操作栈容器来模拟表达式的实际计算过程。当整个表达式对应的指令集合全部执行完毕后，栈容器中最后剩下的元素值便为该表达式的最终求值结果。同时，虚拟机也会将该值作为表达式的计算结果直接返回到用户层。

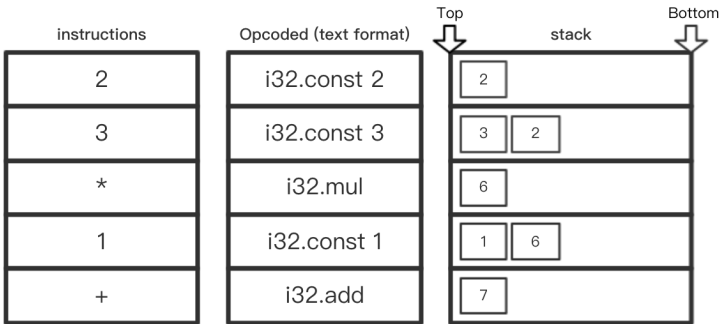


图2-19 RPN表达式“2 3 * 1 +”的执行流程和栈容器的状态变化

基于后缀表示法的 **RPN** 表达式大大降低了堆栈机对表达式求值的难度。同理 **PN** 表达式又称为“前缀表达式”，即在表达式中，运算符被放置在所有关联操作数的最前面。对于该形式的表达式，虚拟机只要从右向左进行解析，即可像 **RPN** 表达式一样，仅通过一个栈容器便可以完成对表达式的求值过程。

2.2.3 Shunting-yard 算法

基于 **RPN** 或 **PN** 表达式的独特性质，堆栈机可以很容易和高效地借助栈容器来解析和执行表达式指令。但实际上，堆栈机上层的开发者还是更习惯使用中缀表示法来表达程序逻辑，因此各种主流的高级编程语言都在其语法标准中规定使用中缀表达式来描述程序逻辑。而这就需要堆栈机的上层编译器能够提供一种方法，通过该方法编译器能够将基于中缀表达式的上层代码“翻译”成 **RPN** 或 **PN** 形式的虚拟机指令代码，最后再交由堆栈机借助栈容器依次执行。

通常来说，在一个表达式中，每一个运算符都有其自己的运算优先级——通过运算符优先级编译器可以知道应该优先计算整个表达式中哪个子表达式的值。除此之外，编译器还需要关注运算符对操作数是否具有“全结合性”。这里的全结合性是指运算符与其两边的操作数是否具有特定的结合律规则。

如果一个运算符具有全结合性，那么在仅由该运算符组成的表达式中，子表达式的具体运算顺序并不会影响到表达式的终值。比如对于一个仅由“+”运算符组成的表达式“1+2+3”，

其括号结合表达式“(1+2)+3”与“1+(2+3)”的终值是一致的，虽然两个子表达式的运算顺序并不相同。因此，我们可以说“+”运算符是具有全结合性的。但如果将上述表达式中的“+”运算符替换成“-”运算符，那么就会发现表达式“(1-2)-3”与“1-(2-3)”的终值并不相同，所以这里“-”运算符并不具有全结合性的。

基于上面提到的表达式运算符特征，堆栈机的上层编译器会使用一种名为“Shunting-yard”的算法来转换输入到编译器内部的上层表达式类型。下面我们以中缀表达式“ $1 + 2 * 3 - 4$ ”为例来介绍 Shunting-yard 算法的具体工作原理。Shunting-yard 算法使用一种特殊的栈结构作为运算符的临时存储容器，同时在输出端使用一种队列结构作为最终转换结果的存储容器。

按照 Shunting-yard 算法的基本工作流程，编译器会从左至右依次扫描中缀表达式中的每一个元素（操作数与运算符）。当遇到的元素类型为操作数时，编译器会直接将该操作数作为输出结果存储到队列容器中。而当遇到的元素类型为运算符时，编译器会按照如下规则来进行处理：如果当前的栈容器中没有任何元素，则将该运算符直接压入栈底；否则，编译器需要将该运算符依次与位于栈顶部分的运算符进行比较。

（1）若当前位于栈顶的运算符其优先级大于该运算符，则编译器直接将位于栈顶的运算符推入输出队列中，然后该运算符继续与下一个栈顶元素进行比较。

（2）若当前位于栈顶的运算符其优先级与该运算符相等，并且位于栈顶的运算符具有全结合性，则编译器直接将位于栈顶的运算符推入输出队列中，然后该运算符继续与下一个栈顶元素进行比较。

（3）若上述规则均不成立，则该运算符会被直接压入当前的栈容器中，成为新的栈顶元素。

当编译器根据上述规则将表达式中的所有元素处理完毕后，会直接将当前用于暂存运算符的堆容器中的所有元素依次推入输出队列中。至此，整个 Shunting-yard 算法便执行完毕了。此时如果直接将输出队列中的元素按照先进先出的顺序导出，那么组合起来便是该中缀表达式对应的 RPN 表达式形式。使用 Shunting-yard 算法将中缀表达式“ $1 + 2 * 3 - 4$ ”转换为 RPN 表达式的整体执行流程如图 2-20 至图 2-28 所示。

Shunting-yard 算法之所以如此命名，是因为其执行过程可以用一个三叉铁路的“枢纽分流图”生动形象地表示出来，如图 2-20 所示。其中左侧表示用于存放算法最终输出结果的队列容器；右侧是输入到算法中的元素，这些元素来源于一个中缀表达式；下方则表示用于暂存运算符的栈容器。首先，算法开始执行并遇到了第一个元素。由于该元素是一个表达式操作数，根据算法规则，该元素会被直接放入左侧的结果输出队列中，如图 2-21 所示。

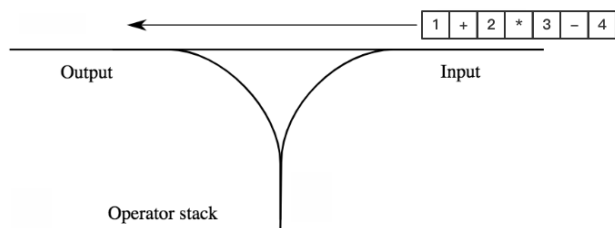


图2-20 Shunting-yard算法的执行流程（1）

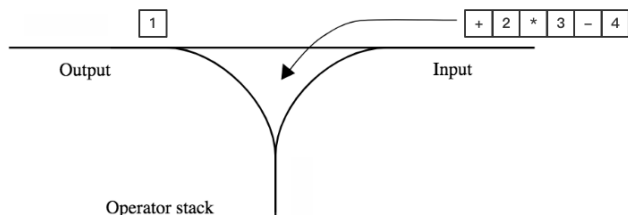


图2-21 Shunting-yard算法的执行流程（2）

接下来遇到的第二个元素是一个“+”运算符。由于此时的运算符栈容器为空，根据算法规则，该元素会被直接压入栈容器的底部，同时成为新的栈顶元素，如图 2-22 所示。

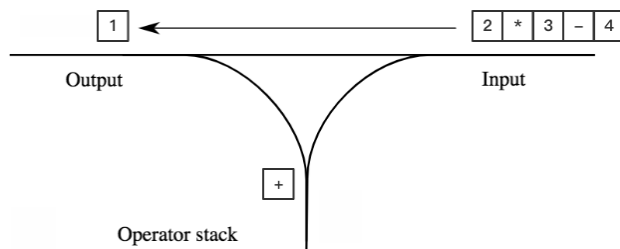


图2-22 Shunting-yard算法的执行流程（3）

接下来的第三个元素是一个操作数，因此会被直接推入结果输出队列的最末端，如图 2-23 所示。

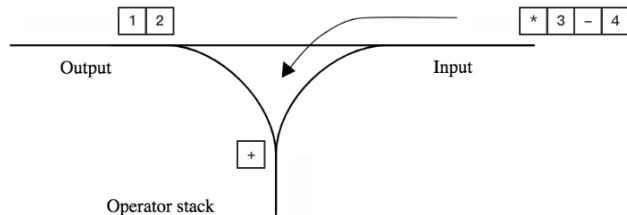


图2-23 Shunting-yard算法的执行流程（4）

第四个元素是一个 “*” 运算符。由于此时栈容器中已经存在 “+” 运算符，因此我们需要根据算法规则来比较该运算符与位于栈顶的 “+” 运算符的运算优先级。比较后我们发现，该运算符并不满足 Shunting-yard 算法的前两条比较规则。因此根据第三条规则，我们将该运算符直接压入当前的运算符栈容器内，此时该元素成为新的栈顶元素，如图 2-24 所示。

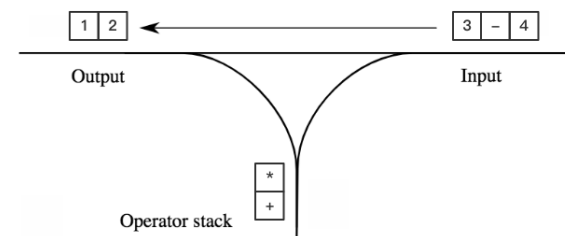


图2-24 Shunting-yard算法的执行流程（5）

第五个元素是一个操作数，因此被直接放入结果输出队列中，如图 2-25 所示。

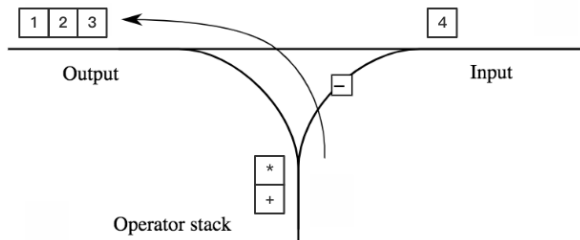


图2-25 Shunting-yard算法的执行流程（6）

第六个元素是一个 “-” 运算符。根据算法规则，我们仍然需要将该运算符与位于栈顶的运算符进行比较。由于 “*” 运算符的运算优先级要高于 “-” 运算符，因此满足算法的第一条规则，直接将 “*” 运算符从栈顶推入结果输出队列中。接下来继续将 “-” 运算符与当前的新栈顶元素 “+” 运算符进行比较。“-” 与 “+” 运算符有着相同的运算优先级，但是 “+” 运算符同时还具有全结合性，因此根据算法的第二条比较规则，“+” 运算符也被推入结果输出队列中。于是，“-” 运算符被暂存到当前的栈容器中成为新的栈顶元素，如图 2-26 所示。

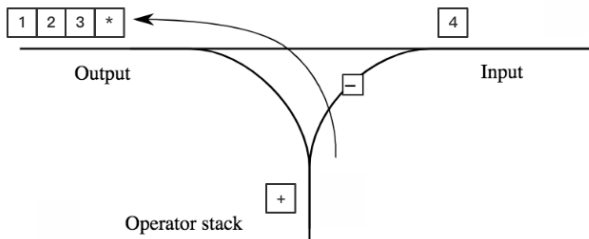


图2-26 Shunting-yard算法的执行流程（7）

第七个元素是一个操作数，我们直接将它放入结果输出队列中，如图 2-27 所示。

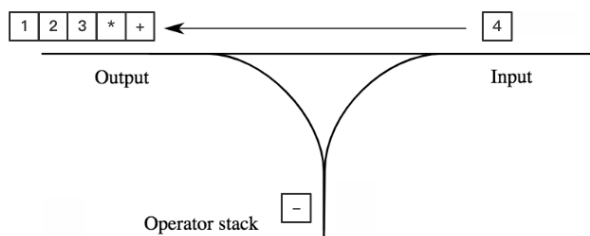


图2-27 Shunting-yard算法的执行流程（8）

此时，输入到算法中的元素已经被全部处理完毕。最后将栈容器中剩余的元素依次推入结果输出队列中即可，如图 2-28 所示。

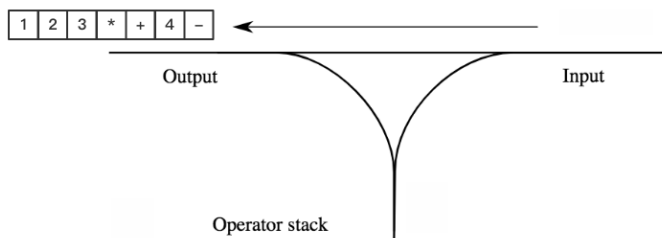


图2-28 Shunting-yard算法的执行流程（9）

从图 2-28 左侧的结果输出队列中可以看到，如果此时将队列容器中的元素按照先进先出的顺序导出并将它们按照字符拼接在一起，便可得到经过算法处理后最终生成的表达式结果，即表达式“1 2 3 * + 4 -”，它便是中缀表达式“ $1 + 2 * 3 - 4$ ”所对应的 RPN 表达式。

以上便是 Shunting-yard 算法在实际应用中的运算流程。关于该算法的 C/C++ 实现细节，可以参考本书配套 Github 仓库“Book-DISO-WebAssembly”中的“ShuntingYard”子目录。

2.2.4 标签与跳转

通常来说，在大多数的虚拟机实现及物理计算机的 CPU 芯片中，最基本的控制流机制便是通过跳转指令来实现的。从操作系统的底层实现方式来看，跳转指令实际上就是将当前操作系统全局指令指针（IP, Instruction Pointer）中存储的指令偏移指向当前内存中一个特定位置，然后操作系统再从指令指针指向的新内存位置开始依次执行内存中存储的指令。

现代计算机的底层硬件大多是基于冯·诺伊曼结构进行设计和制造的。因此，当一个应用

程序在计算机内部运行时，该应用程序所使用的程序指令、数据，以及各类资源实际上都被直接存放在计算机的本地线性内存中。一个完整的应用程序包括了程序在运行中使用到的数据资源、程序代码对应的一系列执行指令，以及一些用于辅助程序正常运行的数据结构和容器。

而实际上，应用程序在运行时是以“Program Memory (PM)”的组成形式被存储到计算机的本地内存中的。PM 描述了一个应用程序运行时的标准内存组成形态，通常一个 PM 由代码段、数据段、BSS 段、堆内存段和栈内存段这几个标准内存段组成，PM 将其所属的内存块分割成几个大小和功能均不同的内存段。与我们之前介绍过的内存分页机制一样，内存分段机制也是一种用于管理计算机线性内存的常用方式。

顾名思义，内存分段管理即将应用程序的运行地址空间划分为若干个段 (Segment) 结构。与内存分页机制的区别是，“段”是一种具有特定信息含义的内存逻辑单位，每一个段都包含一组意义相对完整的信息；而“页”仅表示系统用于内存管理的一种机制，每个页中存储的内容都不尽相同。另外，系统线性内存的分页大小一般都是由操作系统事先确定好的；而内存的分段大小则可以由具体的应用程序来决定，每个应用程序的内存地址空间分段大小都不一样。总体上看，内存分页机制是操作系统用于优化内存利用率的一种底层机制；而内存分段机制可以直接反映出应用程序的内部逻辑，这对于上层开发者更加友好。因此，在现代计算机中，我们通常会把内存分页机制和内存分段机制结合起来使用，这样便形成了“段页式”的内存管理机制。

需要注意的是，从操作系统层面来看，每一个独立的应用程序进程都会对应一个同样完全独立的 PM 内存段，并且 PM 直接对应于应用程序本地的虚拟地址空间 (VAS, Virtual Address Space)，操作系统会给每一个运行中的进程分配同样大小的 VAS（比如对于 32 位操作系统，每个进程可能会被分配 4GB 大小的 VAS）。通过这些完全独立的 VAS，操作系统保证了进程在运行时不会被其他进程随意访问和篡改数据，进而保障了应用程序的安全性。而 VAS 对应的实际物理内存地址，则会由操作系统根据应用的分段信息，以及系统的内存分页信息来进行相应的转换。下面依次介绍 PM 中各个分段的具体作用。

Text（代码段）

代码段又被称为“文本段”。在这块内存中存储的是应用程序的可执行指令。通常来说，这块内存是只读的，并且大小固定。

Data（数据段）

在数据段对应的内存块中，存储的是所有在应用程序源代码中声明并且具有初始化值的全

局变量或静态变量。全局变量是指那些没有定义在函数和块语句中的变量，而静态变量则是指那些使用编程语言关键字显式定义为静态的变量。静态变量可以被定义在函数中，但其定义的具体位置直接影响了被调用的上下文环境。比如在下面 C++源代码中声明的一些变量。

```
// 全局变量
int num = 24;
char string[] = "This is a string."
int add (int a, int b) {
    // 位于函数中的静态变量
    static int amount = 0;
    int result = a + b;
    amount += result;
    return result;
}
```

在应用程序运行前，数据段所对应的 PM 内存块中是没有任何内容的。只有当应用程序启动后，操作系统才会根据其代码段中存放的应用程序代码逻辑，将其中的全局变量和静态变量值复制到数据段中。

BSS (BSS 段, Block Started by Symbol)

BSS 段又被称为“未初始化数据段”。在该内存段中通常存储的是在应用程序源代码中所有初始化为 0，或者没有进行显式初始化的全局变量和静态变量。比如下面代码中声明的静态变量，在应用程序运行时，该变量便会被操作系统分配到应用程序对应 PM 内存块的 BSS 段中。整个 BSS 段的内存空间结构类似于一个栈结构。

```
// 定义了一个没有被显式初始化的静态变量
static int i;
```

Heap (堆内存段)

堆内存段一般从 BSS 段和数据段的末尾开始，并从这里逐渐向内存高地址方向不断增长和扩大。在堆内存段中一般包含了在应用程序源代码中由 malloc 和 new 等内存分配/创建对象函数分配的内存空间，这部分内存空间可以随时通过 free 和 delete 等函数进行释放。比如下面是一段来源于 Eufa 项目的代码，在这段代码中通过 malloc 函数在堆内存段中分配了一块大小固定的内存。

```
...
void* EMSCRIPTEN_KEEPALIVE cache_malloc (size_t size) {
    if (all_cache_used_memory + size > EUFA_CACHE_MAX_MEMORY_SIZE) {
        return NULL;
    }
}
```



```

}
// 在堆内存段中分配一块大小固定的内存
void *ptr = malloc(size + PREFIX_SIZE);
*((size_t*)ptr) = size;
update_zmalloc_stat_alloc(size + PREFIX_SIZE);
return (char*)ptr + PREFIX_SIZE;
}
...

```

Stack（栈内存段）

栈内存段又被称为“调用堆栈”。在这段内存中主要存储了程序在运行过程中产生的函数调用记录。栈内存段通常位于应用程序对应 VAS 的高地址区域。随着函数调用过程的不断增加，栈内存段会逐渐向 VAS 的低地址方向不断增长和扩大。整个栈内存段是由一个后进先出的栈结构容器组成的，在栈容器中存放着由函数调用产生的栈帧对象。

在每一个栈帧对象中都至少包含一个用于返回上一次函数调用位置的内存地址。每当我们在源代码中进行一次子程序（子函数）调用时，操作系统都会在当前应用程序对应的栈内存段的顶端压入一个用于记录该次函数调用位置的栈帧对象。当子程序调用完毕后，操作系统便可以根据栈帧对象中的地址，再次返回到父函数中继续执行剩下的代码。比如将下面给出的这段 JavaScript 代码直接复制到浏览器的控制台中并运行。

```

function recursion () {
  // 递归调用
  recursion();
}

```

代码运行完毕后，我们会发现浏览器抛出了如图 2-29 所示的错误信息。

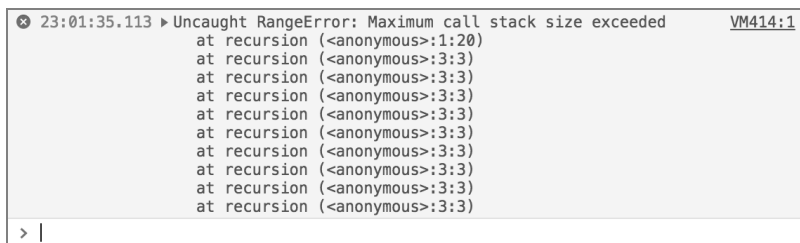


图2-29 浏览器抛出的调用堆栈溢出错误信息

这是由于 `recursion` 函数在代码中不断地对其自身进行递归调用，从而导致调用堆栈溢出而产生错误。每一次进行递归调用时，浏览器都会在当前页面进程对应的调用堆栈结构中压入一

条栈帧记录。随着递归调用的不断进行，调用堆栈也会不断增大，当调用堆栈的大小超过了浏览器的规定大小时，浏览器便会向客户端抛出该错误。

这里需要注意的是，不要混淆“调用堆栈”和“数据堆栈”。调用堆栈主要用来存储每一次调用子程序时的函数返回地址；而数据堆栈则专门用于存储执行指令时需要用到的一系列数据资源。在某些虚拟机实现中，调用堆栈和数据堆栈可能是在同一个栈容器结构中实现的，但两者在概念上的区别仍然十分清晰。

图 2-30 给出的是上面介绍的几个位于 PM 中的常用段结构在 VAS 内部的相对位置。可以看到，堆内存段（Heap）和栈内存段（Stack）占用的 VAS 空间会随着应用程序的运行而不断变化，并且两者在 VAS 空间中占用的地址段并不连续。接下来我们将把目光放回到堆栈机的内部实现原理上。虚拟机在实现的过程中也会参照实际物理机上的应用程序运行逻辑，为运行在虚拟机上的应用程序模拟出同样的一套位于其内存空间的“段结构”。

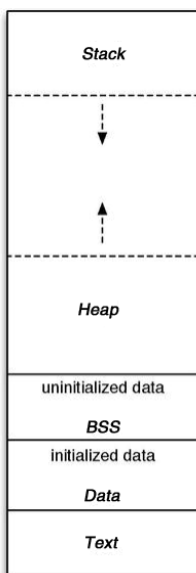


图2-30 PM中常用段结构的所在相对位置

当我们在应用程序源代码中使用 if 或 while 等高级编程语言中的条件/循环语句时，包含在这些语句块中的代码只有在符合特定条件的情况下才会执行。而实际上，当这些代码被运行在堆栈式虚拟机上时，源代码中的条件语句都会被直接转换成虚拟机底层的跳转指令。因为在大多数情况下，堆栈机本身并没有结构化的控制流机制，被保存在应用程序虚拟地址空间“代码段”中的指令只能够逐条按次序被堆栈机线性执行。

```
...
label(main)
    ; 一些指令
    jmp(skip)
label(garbage)
    ; 一些指令
label(skip)
    ; 一些指令
...
```

通过上面的伪代码，我们模拟了一个堆栈机应用的 VAS 代码段中虚拟指令的大致结构。可以看到，代码段中的指令被 `label` 标签分割成不同的代码片段，每一个代码片段都对应着一个独立的代码逻辑单元。在这段代码中，`label` 指令的作用是为代码段中某个位置的代码偏移设置一个标签，这样虚拟机便可以通过调用 `jmp` 指令来让当前堆栈机的全局指令指针能够在各个代码片段之间进行跳转。

比如在上面的伪代码中，堆栈机首先会从 `main` 标签处开始执行指令，而在执行到 `garbage` 标签内的代码前，`jmp(skip)` 指令便会直接将当前堆栈机的全局指令指针指向 `skip` 标签处的代码，接下来堆栈机便会从 `skip` 标签处继续执行指令。这种指令跳转方式我们称之为“无条件跳转”。通过结合这种指令指针的跳转方式，以及存放在栈容器中的数据，堆栈机便可以使用线性指令来模拟诸如 `if` 和 `while` 等复杂条件跳转语句的执行流程。注：上面代码中给出的 `label` 标签及 `jmp` 指令均为伪指令，这里只用于模拟代码段中底层虚拟指令的结构。关于具体的虚拟指令集，还需要根据堆栈机的具体实现来确定。

2.2.5 条件语句

下面我们将介绍几个常用于模拟条件跳转的虚拟指令。同样的，这些指令均为伪指令，它们并不代表任何具体堆栈机实现上的虚拟指令（OpCode），这里只方便用于描述堆栈机内部的指令运行机制。

jz 指令（为 0 则跳转）

该指令首先会从当前栈容器中弹出一个元素，即位于栈顶的元素。若该元素的值为 0，则会发生指令指针跳转。伪代码示例如下。

```
...
label(loop)
    jz(end)
```

```
; 一些指令
jmp(loop)
label(end)
nop
...
```

上面这段指令代码模拟了高级编程语言中的 `do...while` 循环语句。堆栈机会从 `loop` 标签处的指令开始执行。在执行第一条 `jz(end)` 指令时，堆栈机会取出当前栈容器中位于栈顶的元素，然后判断该元素的值是否为 0。如果为 0，则指令指针会直接跳转到 `end` 标签处，跳出 `loop` 标签内的指令，否则会继续执行 `loop` 标签内剩下的指令。其中标签内的最后一条 `jmp(loop)` 指令会将指令指针重新移动到 `loop` 标签处，并开始下一次循环过程。在这里我们可以随时通过修改位于栈顶的元素来决定是否跳出该循环结构。

实际上，在不同情况下，`jz` 指令的具体实现方式是不同的。这里假设堆栈机在执行该指令时会将栈容器顶部的元素弹出，但是在某些情况下，这可能不是我们想要的结果（被该指令比较的栈顶元素将丢失）。

je 指令（标志位跳转）

该指令会判断特定的标志位寄存器的值是否为 0。若条件成立，则会进行相应的指令跳转。伪代码示例如下。

```
...
cmp eax, 5
je if_true
; 一些指令
label(if_true)
; 一些指令
label(end)
nop
...
```

在这段代码中，我们首先通过 `cmp` 指令来判断 `eax` 寄存器中存放的值是否为 5。若条件成立，则 `je` 指令对应的标志位寄存器便会被重置为 0。这样当堆栈机在接下来的指令执行过程中遇到 `je` 指令时，便会进行相应的指令跳转。

在上面代码中，若 `eax` 寄存器中的值为 5，指令指针便会跳转到 `if_true` 标签处。通过结合 `je` 和 `cmp` 指令，我们可以模拟出高级编程语言中的 `if` 条件语句。

2.2.6 子程序调用

为了能够在堆栈机的底层虚拟指令中更加清晰地描述高级编程语言中的函数特性，在堆栈机中又引入了“子程序”的概念。子程序即通过将部分堆栈机虚拟指令封装成类似函数的形式，使得这些指令簇可以在代码中被反复调用。这里为了能够完整地封装并调用一个子程序，我们还需要了解如下两种虚拟指令。

ret 指令（从子程序返回）

该指令要求堆栈机从当前应用程序 PM 内的调用堆栈中弹出位于栈顶的调用栈帧，并根据栈帧中存放的指令返回地址将控制权返回给子程序的上层调用者。在调用栈帧中存放的是当前堆栈机正在运行子程序的上层调用者的指令返回地址，堆栈机会通过直接改变全局指令指针的方式来将控制权返回。该指令必须且只能放在每个子程序结构的末尾。

call 指令（调用子程序）

通过该指令堆栈机可以直接调用一个定义完整的子程序。

接下来，我们通过结合这两个与子程序相关的指令，编写一段简单的伪代码来了解子程序的具体组成结构及调用方法。伪代码片段如下。

```
...
jmp(start)

; 定义一个子程序
label(printAll)
  label(loop)
    ; 从栈容器顶部弹出一个元素，若该元素的值为 0，则跳转到 end 标签处
    jz(end)
    ; 弹出位于栈顶的元素并向上层环境打印
    echo
    jmp(loop)
  label(end)
    nop
; 返回到上层调用者
ret

label(start)
; 向栈容器中写入数据
```

```
push 0
push 1
push 2
push 3
; 调用子程序
call(printAll) ; 3 2 1
...
```

在这段伪代码中，首先向数据栈容器中压入了四个值，然后通过调用一个子程序将栈容器中位于栈顶的前三个值分别打印到了堆栈机的上层环境中。如伪代码所示，我们可以直接通过 `label` 标签指令来定义一个子程序。并且在子程序中所有指令的最后，需要通过 `ret` 指令将程序的控制权返回给子程序的上层调用者，堆栈机也正是通过该指令来判断一个使用 `label` 指令定义的代码段是否是子程序类型的。

从另一个方面来看，我们可以将子程序理解为高级编程语言中的函数。但是在大多数编程语言中，一个函数通常只能有一个返回值。由于堆栈机具有基于栈容器来进行数据交换和传递的特性，这使得实际上一个子程序在调用完毕后可以向上层调用者返回多个结果值。子程序被调用时，可以从当前的栈容器中直接获取任意数量的操作数来使用。同样的，子程序执行完毕后，也可以在栈容器中存放任意数量的元素作为其调用后的返回值。一旦子程序执行完毕，在子程序上层调用者的环境中便可以直接读取和使用这些存放在栈容器中的元素。

2.2.7 变量

到目前为止，我们介绍过的所有伪代码示例中的数据元素都是被直接存放在数据栈容器中的。由于栈结构本身具有后进先出的特性，因此我们无法以较低的操作成本对位于栈容器底部的数据元素随意地进行读取和修改操作。换句话说，数据栈容器本身并不是专门用来存储长生命周期数据的，它主要用于临时存储那些正处于数据运算和交换过程中的数据元素。

而“变量”是专门用于存储数据的一种方式。从底层结构来看，变量与标签都是指内存中某个位置的名称，其中标签主要用于标记代码段中的内存偏移地址；而变量则是指内存中某个数据容器的名称。从更高的层次来看，一个变量就是一个存储在内存中的数据项，每一个变量都包含一个与之相对应的值，并且该值可以随时间和程序的运行而发生改变。变量的值可以通过 `set_val` 和 `read_val` 指令来进行写入和读取操作。

`set_val` 指令（写入变量）

通过该指令，可以对一个变量写入特定的值。堆栈机在执行该指令时会自动在内存中为变量值动态分配空间，这个过程并不需要通过指令来明确指定内存的分配位置与大小。

read_val 指令（读取变量）

我们可以通过该指令读取一个变量的值。该指令会自动查找并返回位于内存中符合特定变量名称的变量值，该值返回后会被直接压入数据栈容器内作为新的栈顶元素，而原先存储在内存中的变量值则不会发生改变。

通过上面介绍的两个指令，我们便可以完成对变量的赋值和读取操作。伪代码示例如下。

```
...
jmp(start)

label(start)
; 向栈容器中写入数据
push 1
; 从栈容器顶部弹出一个元素，并将该元素的值赋值给变量 i
set_val(i)
; 向栈容器中压入新的数据
push 2
; 从内存中读取变量 i 的值并压入数据栈容器中
read_val(i)
; 从栈容器中弹出两个元素，计算其“和”后再将结果压入栈容器中
add
; 将栈容器中的计算结果再次赋值给变量 i
set_val(i) ; 3
...
```

实际上，这里在代码中创建的所有变量其作用范围都是全局性的，堆栈机可以在整个伪代码段的任何位置使用这些变量。而一旦在代码中进行了子程序调用，那么位于子程序外部的所有变量便都无法在子程序内部使用了。对于这些在子程序外部创建的变量，堆栈机会在执行子程序时使用一种不同的方式来进行转存和处理。总的来说，变量只能在同一层级的作用域内进行使用，不同的（子）程序之间无法进行共享。这在某种程度上减少了变量可能产生的副作用。

2.2.8 栈帧

在前面的章节中我们介绍过，位于应用程序 PM 栈内存段中的调用堆栈主要用来存放子程序调用时产生的栈帧。在每一个栈帧结构中都存放了用于将控制权返回给上层调用者，即上一次函数调用位置的内存指令返回地址。而实际上，栈帧中除了存放用于返回上层调用位置的指针地址，还存放了上层作用域内使用到的所有局部变量值，以及其他相关的环境信息。

当子程序执行完毕时，堆栈机会根据调用堆栈中位于栈顶的栈帧信息，将当前的全局指令

指针设置到上层调用者处。同时堆栈机还会对上层调用环境内部的所有可用局部变量再次进行初始化，并恢复到子程序调用前的值。

2.2.9 堆

为了能够在堆栈机中更加高效地进行数据读写操作，这里又引入了“堆”的概念。“堆”即应用程序 PM 堆内存段对应的线性内存块。如果只是单纯地将数据栈容器作为数据的存储容器，则会出现无法低成本地读取位于栈顶以下的数据，以及数据的存放与读取顺序相反等问题。在某些情况下，这些问题会导致编译器生成大段冗余复杂的虚拟指令，同时堆栈机中数据栈容器的定位也会变得模糊。因此，在堆中进行数据读写是堆栈机支持的另一种数据存储方式。

堆作为数据存储的主要方式之一，与变量相比，在堆中进行数据存储需要我们手动指定待存放数据的起始内存地址，以及对应的数据值，堆栈机并不会自动管理堆中的数据。因此，当这些数据不再需要时，也必须手动来释放数据占用的内存块，以防止发生内存泄漏的问题。

那么，我们应该在什么样的情况下使用堆来作为数据的存储方式呢？在大多数情况下，基于栈容器中的数据读取与写入速度会比使用堆的速度快。因此，如果需要分配一大块内存作为数据的存储区域（如大数组与结构体），并且需要在应用程序运行期间长时间地保持这些数据（如全局变量），那么在堆内存中分配和存储这些数据可能会更加合适。

实际上，上层高级编程语言中类似 `malloc` 的堆内存分配函数会被编译器编译为堆栈机的底层虚拟指令。下面我们来介绍两个堆栈机伪指令，它们分别用于对堆内存中的数据进行写入和读取操作。

load 指令（从堆内存中读取某一位置的数据）

该指令主要用于从堆内存中的某一特定位置读取数据，读取出的数据会被直接压入当前的数据栈容器中。该指令在执行时接收一个操作数，即堆内存中某一位置的线性偏移量。

store 指令（向堆内存中的某一位置写入数据）

该指令主要用于向堆内存中的某一特定位置写入数据。该指令在执行时需要接收两个操作数，其中一个为堆内存中某一位置的线性偏移量；另一个为需要写入该堆内存位置的具体数据值。

接下来，我们通过下面给出的伪代码片段来了解这两个指令的具体使用方法。

```
...  
jmp(start)
```



```

label(start)
  // 在堆内存中的位置 0 写入数据 1
  store 0 1
  // 在堆内存中的位置 1 写入数据 1
  store 1 1
  // 该 add 指令会直接计算堆内存中位置 0 和位置 1 的数据值之和，并将计算结果写入位置 2
  add 0 1 2
  // 从堆内存中的位置 2 读取数据，并压入栈容器中
  load 2 ; 2
...

```

在这段伪代码中，首先通过 `store` 指令在堆内存的位置 0 和位置 1 处分别同时写入数据值 1。接下来，通过一个特殊的 `add` 指令来计算堆内存中位置 0 和位置 1 的数据值之和，并将计算结果直接写入堆内存中的位置 2 处。最后，再通过 `load` 指令将堆内存中位置 2 处的数据值读取出来并压入当前的数据栈容器顶部。这里提到的堆内存位置 0、1 和 2 是指相对于堆内存基准位置的偏移位置（量）。

至此，我们便基本介绍完了堆栈机的相关组成和实现原理及虚拟指令运行逻辑。但实际上，现阶段 WebAssembly 标准抽象出的“Wasm 抽象虚拟机”并不是一个完整的堆栈机形态。Wasm 抽象虚拟机只借鉴了堆栈机中基于栈容器的结构化控制流，以及基于栈本身的数据结构化应用等部分特性。而对于诸如“子程序多返回值”等特性，则会在 Post-MVP 标准中再做进一步的评估与实现。需要再次强调的是，所谓的“Wasm 抽象虚拟机”是指从 WebAssembly 标准中对其虚拟指令形式和执行方式的设计上来看是符合堆栈机的部分特性的，但这并不意味着例如 V8 等 Wasm 执行引擎会在具体实现时真正采用栈结构的容器，这个是需要注意区分的。

堆栈机模型除其本身具有实现较为简单、较短的虚拟指令集使其生成的二进制 OpCode 模块文件密度适中，以及不依赖特定寄存器使其可移植性较好等特性外，对于 WebAssembly 来说，另一个巨大的优势就是基于栈容器的结构化控制流和数据流，使得 Wasm 引擎对 Wasm 模块的可用性检查变得十分简单和高效。Wasm 模块的可用性检查包括对模块的完整性、可用性，以及内部 Wasm 虚拟指令（OpCode）是否符合相应语法标准等方面进行的一系列检查。这些检查对于防止 Wasm 模块在实例化和使用过程中出现标准中未定义的行为有着极其重要的作用。

2.3 类型检查

浏览器在加载和实例化一个 Wasm 模块之前，所需要进行的一个最重要的步骤就是对模块进行可用性检查。在这一节中，我们将介绍 Wasm 模块可用性检查中最重要的一部分，即类型

检查。特别提示：本节内容中出现的所有用于介绍原理性知识的虚拟机指令代码都来自真实的已经发布的 WebAssembly 最小可用版本（MVP）标准。这里可以结合上一节中介绍的堆栈机常用伪指令及指令的基本运行原理，来理解这些标准 Wasm 虚拟指令的功能和工作原理。

2.3.1 数据指令类型

由于 Wasm 抽象虚拟机本身也是基于堆栈机模型来实现的，因此对于如下一段标准的 Wasm 虚拟指令代码，指令操作数的传递和数据交换过程也是基于数据栈容器来进行的。

```
i32.const 10  
i32.const 20  
i32.add
```

在这段 Wasm 虚拟指令代码中一共有三条虚拟指令，每一条指令在表达形式上都指明了其在执行过程中的操作数类型，以及指令本身的功能。在 WebAssembly 现阶段的 MVP 标准中规定，指令操作数被分为“i32”“i64”“f32”和“f64”四种类型。每种类型标识符的第一个字母“i”和“f”分别指定了操作数的类型为整型和浮点型，而字母后的数字“32”和“64”又分别指定了操作数的最大长度为 32 位和 64 位。在操作数类型标识符后面的便是指令本身的功能标识符，操作数标识符和功能标识符通过一个“.”号来进行连接。

在上述指令片段中，首先通过 i32.const 指令向当前的数据栈容器中依次压入了两个 32 位的整型数“10”和“20”。接下来，i32.add 指令直接从当前的数据栈容器中弹出位于栈顶的两个元素，并在内存中计算这两个元素的数字值之和，最后再将计算结果压回栈容器中。各条虚拟指令执行时的栈容器状态变化如图 2-31 所示，这里假设栈容器的初始状态为空。

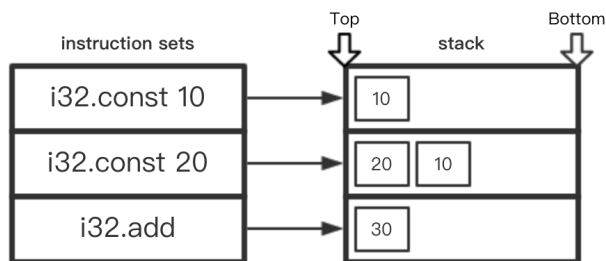


图2-31 上述Wasm虚拟指令段执行时的栈容器状态变化

可以看到，在 Wasm 虚拟指令的执行过程中，所有指令的运行参数都是符合特定的类型要求的。比如 i32.add 指令在执行时从当前栈容器顶部获取到的“10”和“20”均符合参数 i32 的要求，即可以作为 32 位长度的整型数来处理。同理，若在 i32.add 指令执行前，栈容器中当

前存放的数据其类型或个数不符合参数要求，如图 2-32 所示的栈容器状态，则相应的虚拟指令便会执行失败。

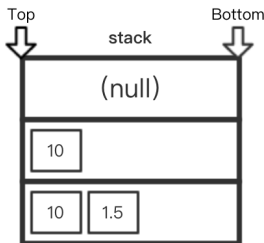


图2-32 无效的数据栈容器状态

图 2-32 中的三种无效的栈容器状态分别是：栈容器为空（null）、栈容器中只存放了一个操作数，以及栈容器中存放的两个操作数类型不符（其中一个为浮点数类型）。实际上，浏览器在 Wasm 模块实例化前就需要完成对其内部指令逻辑的操作数类型判断，但此时我们并不知道模块在实际运行过程中与数据堆栈交互时存放的具体元素值是什么。比如在如下的指令代码中，操作数完全来自用户的实际输入，因此我们并不能判断出指令在执行时存入数据栈容器的具体数值是什么。

```
get_local $0
get_local $1
i32.add
```

在这段指令代码中，我们首先通过两条 `get_local` 指令连续获取了两个名称分别为“\$0”和“\$1”的本地变量值，并将它们依次压入当前的数据栈容器中。但实际上，我们并不清楚这两个本地变量的具体值是什么。`get_local` 指令一般用来获取函数的形参值，函数的形参会在 Wasm 模块实例化时被映射为函数本地变量空间中的变量。而正是由于这些形参的具体值均来自外部用户的输入，因此我们无法在验证模块时确定传入函数的参数值，进而无法判断参数的类型是否正确，即无法确定放入栈容器中的值其具体数据类型。

换个思路来看，虽然我们无法得知栈容器中存放的具体操作数值，但是由于 Wasm 标准中所有用于标记数据的指令都带有明确的类型标识，因此我们可以直接通过判断数据转移路径上的所有类型节点是否正确，进而判断模块是否符合 WebAssembly 的类型验证标准。为此，我们需要使用“类型栈容器”来代替“数据栈容器”对指令代码进行类型分析。

```
(func $add (; 1 ;) (param $0 i32) (result i32)
  (i32.add
    (get_local $0)
```

```

    (i32.const 1)
  )
)

```

上面给出的是一段基于结构化虚拟指令表达的 Wasm 模块函数定义过程。所谓的结构化指令表达可以理解为将 Wasm 模块的二进制内容通过带有控制结构和语法结构的虚拟指令代码以具有一定人类可读性的方式表达出来。这种表达方式我们又称之为“Wasm 可读文本格式 (WAT, WebAssembly Text Format)”，简称为“WAT”格式。WAT 将 Wasm 的标准虚拟指令以“S 表达式”的方式进行组织和表达。关于 WAT 的进一步内容我们会放在下一章中进行介绍。

回过头来看，在上述函数定义过程中，我们首先通过 `get_local $0` 指令获取了一个名为“\$0”的本地变量值。可以看到，通过该指令获取的变量名“\$0”实际上来自函数 `$add` 的形参，并且该参数也在函数的声明过程中通过“`param $0 i32`”的方式被标记成了“i32”类型，即 32 位整型。函数会从用户那里接收到一个具体的参数值，该参数值会作为函数的实参进入函数内部的数据处理逻辑中。从宏观角度来看，在函数内部定义的每一个数据操作指令都会对该参数的数据形态（值和类型）做出处理，而对于使用这些指令处理后产生的新数据值则可以通过“类型堆栈”来进行跟踪，如图 2-33 所示。

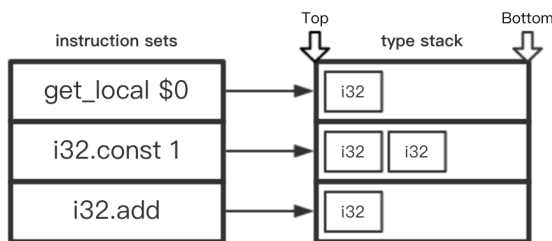


图2-33 上述Wasm函数执行时对应的“类型堆栈”状态

实际上，我们可以将每一个数据操作指令都看作一个数据状态的“处理机”。这些处理机从外部接收数量和类型固定的参数作为输入，在对输入数据经过一定处理后，又会产生固定数量和类型的数据作为输出。而对于数据在经过指令处理的前后状态，我们都可以通过类型栈容器来进行跟踪和判断。

2.3.2 基本流程控制

与通常的堆栈机模型一样，WebAssembly 抽象虚拟机也同样具有用于流程控制的相关指令。比如 `block` 指令用来声明一个包含有若干条虚拟指令的指令段。该指令的功能与上一节中介绍的 `label` 指令类似，其具体用法如下。

```
block $main
;; 一些指令
end
```

在上面的指令代码中，我们通过 `block` 指令声明了一个名为“\$main”的指令段。`end` 指令用于标记块级作用域的结尾，因此这里使用它来标记\$main 指令段的结束位置。而所有放置于 `block` 与 `end` 中间的指令，均属于该指令段。`block` 指令定义了最基本的指令块结构，在此基础上，我们便可以通过 `br` 无条件跳转指令、`br_if` 及 `br_table` 等条件跳转指令，让模块的全局指令指针在各个指令块之间进行跳转。

br 指令

`br` 指令会中断指令段的执行，即该指令会让模块的指令指针跳出特定的指令段，并继续执行位于指令段后面的指令代码。该指令的具体用法如下。可以看到，通过 `br` 指令，我们便可以无条件地直接中断\$main 指令段的当前执行流程，并继续执行位于该指令段后面的指令代码。

```
block $main
...
br $main    ;; 中断$main 指令段执行
nop         ;; 不会被执行到的指令代码
end
;; 一些指令
```

br_if 指令

`br_if` 指令的功能与 `br` 指令类似，两者均可以直接中断一个特定指令段的执行流程。但不同的是，`br_if` 是一种条件跳转指令，即该指令在执行时需要满足特定的数据栈容器状态才可以产生跳转效果。`br_if` 指令在执行时，会首先从当前的数据栈容器中取出位于栈顶的元素并进行值判断。若该元素的值不为 0，则会进行相应的中断跳转；否则会继续执行该指令段后面的指令代码。比如在下面给出的示例代码中，我们首先通过 `get_local` 指令获取一个名为“\$var”的本地变量值并将其压入当前的数据栈容器中。而当 `br_if` 指令执行时，堆栈机会通过判断当前数据栈容器的栈顶元素值是否为 0，来决定是否需要进行相应的中断跳转操作。

```
block $main
  get_local $var
  br_if $main
  ;; 如果本地变量$var 的值为 0，则会继续执行此处的指令代码
nop
end
;; 如果本地变量$var 的值不为 0，则会中断$main 指令段并执行其后的指令代码
```

br_table 指令

br_table 也是一种条件跳转指令。但不同的是，**br_table** 在其内部维护了一个存储众多指令段标签的标签向量。一个完整的 **br_table** 指令由一个初始值均为 0 的标签向量、一个默认标签和一个索引操作数组成。该指令在执行时，堆栈机会根据索引操作数在标签向量中查找对应索引位置的标签，并进行标签中断与相应的指令跳转。但如果索引操作数超过了标签向量的长度，则堆栈机会将全局指令指针直接移动到 **br_table** 指令给定的默认标签位置。

上面介绍的 **br**、**br_if** 和 **br_table** 指令便是 WebAssembly 结构化控制体系中最基本的三种底层流程控制指令。通过这三种指令，便可以模拟高级编程语言中的循环、选择和条件等常用的结构化控制语句。但事实上，WebAssembly 标准中还规定了一些可以直接用于模拟循环、选择和条件三种语句的虚拟指令。在不同的情况下，可以选择使用底层流程控制指令，或者直接使用 Wasm 标准中提供的结构化控制指令来模拟上层结构化控制语句，两者有其各自的使用场景和优势。特别是在某些情况下，编译器会根据不同的代码逻辑情况来使用不同类型的指令对代码进行优化，从而生成体积更小的二进制代码。

if 指令

使用 **if** 指令可以直接模拟高级编程语言中的条件分支语句。该指令的具体用法如下。首先通过 **get_local** 指令将本地变量 **\$var** 的值压入栈容器的顶部。**if** 指令在执行时，会将位于栈容器顶部的元素弹出并进行判断，若该元素的值不为 0，则会直接执行 **if** 指令段内的指令代码；否则会执行 **else** 指令段中的指令代码。**if** 也是一种“块”指令，因此需要通过 **end** 指令来标记 **if** 指令段的结束位置。

```
get_local $var
if
    ;; 如果本地变量$var 的值不为 0，则会执行此处的指令代码
else
    ;; 否则会执行此处的指令代码
end
```

对于上述指令段的逻辑，我们也可以使用 **br** 和 **br_if** 指令来进行改写，改写后的指令段如下。可以看到，在功能及内部逻辑保持不变的情况下，我们通过一个嵌套的且名字不同的 **block** 指令块将条件语句对应的两个执行分支进行了分离。在这里，**br_if** 指令控制着整个指令段的逻辑选择过程，即上述 **if** 指令本身的条件判断功能；而 **br** 指令则负责创建一个选择分支结构。当一个分支执行完毕后，**br** 指令会通过中断最外层 **\$main** 指令段的方式来退出整个 **if** 结构。

```
block $main
```

```

block $true
  get_local $var
  br_if $true
  ;; 如果本地变量$var 的值为 0，则会执行此处的指令代码
  br $main
end
;; 否则会执行此处的指令代码
end

```

与高级编程语言一样，这里的 if 指令也可以仅保留一个分支，其 else 分支可以被省略。由于 if 指令与 block 指令一样，也是一种会生成块级指令段结构的指令，因此我们同样可以在 if 指令生成的块结构中，随时使用 br 和 br_if 等流程控制指令来中断当前 if 结构的执行流程。代码示例如下。

```

get_local $var
if $main
  br $main
  ;; 此处的指令无法执行
end
;; 中断后继续执行此处的指令代码

```

在这段指令代码中，我们为 if 块结构标记了一个名为“\$main”的别名标签，利用该标签，便可以随时通过 br 和 br_if 指令将指令执行流程从当前的 if 块结构退出到外层结构中。

loop 指令

loop 指令用来标记一个循环结构。之所以说用来“标记”一个循环结构，是因为 loop 指令本身并不会完全构建一个可以自动进行指令回溯的循环结构。与大多数高级编程语言不同的是，WebAssembly 标准中规定控制流程不能够自动进行指令回溯，即在一个循环结构中，需要显式告知堆栈机应该在何时将当前指令指针回溯到循环体的顶部，以便开始下一轮循环。没有包含显式指令回溯操作的 loop 循环结构就相当于一个最基本的 block 结构块，其中的指令代码只会按照正向执行的方式从上到下执行一次，比如下面这段指令代码。

```

loop $main
  nop
  ;; 这里的指令代码会被执行一次
end
;; 这里的指令代码也会被执行一次

```

在这段代码中，我们通过 loop 指令创建了一个名为“\$main”的循环结构。但在块结构中却并没有显式告知堆栈机需要在何时将循环结构内的指令指针向上回溯，以便进行下一轮循环。

因此，这段指令代码实际上只会从上到下按顺序执行一次。此时 `loop` 指令的作用与 `block` 指令一样，只是标记了一个循环结构，而指令代码的实际执行逻辑没有发生任何变化。

为了能够真正产生循环结构的多次迭代效果，我们可以通过 `br`、`br_if` 和 `br_table` 指令来显式告知堆栈机应该在何时将当前 `loop` 块结构中的指令进行回溯，并开始新一轮循环。在下面给出的一段示例代码中，首先通过 `loop` 指令构建了一个名为“`$continue`”的循环结构。在块结构内部，我们会根据实际的业务流程来不断修改变量 `$var` 的值，而该变量同时也会作为循环结构的迭代标识控制着整个循环结构的迭代次数。在循环结构的末尾，我们通过 `get_local` 指令将变量 `$var` 的值压入数据栈容器中。接下来的 `br_if` 指令会判断位于栈顶的元素值是否为 0，若不为 0，则堆栈机会直接将循环结构回溯到 `loop` 结构块的第一行指令，并开始一轮新的循环；否则，堆栈机会继续执行后面的指令，并按顺序退出循环结构。

```
loop $continue
  ;; 根据业务逻辑修改$var 变量的值
  get_local $var
  br_if $continue
  ...
end
```

总的来说，`br`、`br_if` 和 `br_table` 指令在配合不同逻辑控制指令使用时会有不同的作用和效果。对于 `if` 及普通的 `block` 指令块来说，这三种指令的主要作用是中断当前指令块的执行流程，并将指令指针移动到外层指令块中继续执行；而对于 `loop` 指令块来说，这三种指令则主要用于控制循环的迭代过程，可以用于模拟高级编程语言中的 `continue` 及 `break` 关键字。

与数据指令一样，对于使用 `if`、`block` 和 `loop` 这三种基于块结构的逻辑控制指令构建的指令段，浏览器也需要在模块被加载和运行前对它们进行严格的类型检查。而检查遵循的主要原则是：经过逻辑控制指令段后的数据栈容器大小不能小于其初始大小。这意味着我们不需要知道数据栈容器中的所有数据状态，即可完成对这些块结构指令的类型校验过程。下面给出一个基于 `block` 指令构建的指令段。由于该指令段不符合上面提到的类型检查原则，因此其所对应的 Wasm 模块无法被浏览器正常加载和实例化。

```
i32.const 1
i32.const 2
block $main
  i32.add ;; 无效的指令
end
```

在这段指令代码中，首先通过 `i32.const` 指令向数据栈容器中压入了两个值分别为“1”和

“2”的元素，此时栈容器中有且仅有这两个元素。然后通过 `block` 指令构建了一个名为“\$main”的块结构指令段，在这个指令段中直接调用了 `i32.add` 指令。我们之前介绍过，`i32.add` 指令会直接从当前的栈容器中取出位于栈顶的两个元素，然后在内存中计算这两个元素的和，最后再将计算结果压入栈容器中。但在这段代码中，`i32.add` 指令却无法被正确执行。这是因为 `block` 块结构内部消耗了在进入块结构之前栈容器内保存的初始元素。如果块结构中的指令能够被执行，那么当指令指针再次返回到块结构外部时，栈容器的大小将会小于其初始大小，即进入块结构前的栈容器大小，这样便违背了块结构的类型检查原则。

2.3.3 基于表达式的控制流

除上面介绍的基于结构化指令的控制流外，在 WebAssembly 标准中还制定了一系列基于表达式的控制流指令。所谓的基于表达式可以将其理解为随着结构化指令段的执行，指令段会向外层块结构中返回一些特定的数据，而这些数据最终会被压入数据栈容器中作为结构化指令段的执行结果。我们可以这样来进一步理解基于表达式的控制流指令：在高级编程语言中，可以将最基本的条件控制语句（`if... else...`）理解为一种基于结构化指令的控制流方式；而利用三元运算符（`... ? ... : ...`），同样也可以实现与条件控制语句相同的控制逻辑。但不同的是，通过三元运算符构建的是各种各样的表达式而非控制结构，并且表达式在执行完毕后还会向用户环境返回表达式的运算结果。

从宏观角度来看，基于表达式的控制流指令即在结构化控制流的基础上，向数据栈容器中返回一个作为其执行结果的数据值。因此，只要直接对结构化控制流指令稍做修改，便可将其变为基于表达式的控制流指令。在这里，我们需要为 `if`、`block` 和 `loop` 这三种结构化控制流指令加上一个用于表明其返回值类型的“类型签名”。类型签名会明确告知堆栈机当块结构中的指令全部执行完毕时，结构化控制流需要向栈容器中存放的返回值类型。而只有当返回值的具体类型与类型签名保持严格一致后，这段基于表达式的控制流指令才能够完全通过虚拟机对 Wasm 模块的类型验证过程，并正常加载和实例化。在如下指令段中，我们只需要对 `if` 逻辑控制指令稍做修改，便可以将其变成一种基于表达式的控制流指令。

```
get_local $var
if $main i32
  i32.const 2
else
  i32.const 3
end
;; 根据$var 变量值的不同，当前栈容器栈顶元素的值可能为 2 或 3
```

可以看到，在“if \$main i32”这句指令中，我们为 if 指令段设置了一个名为“\$main”的标签，同时还标记了该段结构化控制流需要向栈容器中返回的结果值为 i32 类型，即一个 32 位整数。在接下来的指令执行过程中，if 指令段会根据 \$var 变量的不同值向栈容器中压入不同的整数作为其返回结果。如果 \$var 变量的值不为 0，则会执行“i32.const 2”指令将数字“2”压入栈容器中；否则，数字“3”会被压入栈容器中成为栈顶元素。与 if 指令段一样，我们也可以为 block 和 loop 指令段以同样的方式标记其控制流的返回值类型签名。比如下面给出的是 block 指令段的类型签名示例。

```
block $exit i32
...
;; 向栈容器中压入一个整数 5
i32.const 5
;; 当前栈容器中位于栈顶的元素与该指令段的类型签名一致，因此可以正常中断
br $exit
end
;; 当前栈容器顶部的元素值为 5
```

我们为 block 指令段标记了一个 32 位整数的返回值类型签名。即当整个指令段中的指令执行完毕时，指令段需要向栈容器返回一个 i32 类型的整数作为其返回值。所谓的“指令段执行完毕”，对应于两种常见方式：第一种方式是指令段从上至下按顺序执行，最后经过 end 指令结束并退出；第二种方式则是通过诸如 br、br_if、br_table 等流程中断指令主动从当前指令段的执行流程中退出。但无论指令段以哪种方式执行完毕，只要在代码中为这些结构化控制指令标记了返回值类型签名，在退出指令段前就必须保证栈容器的栈顶元素类型符合该返回值类型。比如下面这段指令代码便是由于指令段的具体返回值类型与其类型签名不一致，从而导致指令无法被正常执行。

```
block $exit f32
...
;; 向栈容器中压入一个整数 5
i32.const 5
;; 由于当前栈容器顶部的元素不是浮点类型，因此该中断指令无法执行
br $exit
end
```

还记得我们之前介绍过的 loop 指令吗？当在一个由 loop 指令构成的指令段中使用中断指令时，其作用并不是中断当前指令段的执行流程，而是将指令指针向上回溯并开始进入下一轮的指令迭代过程。因此，在一个基于 loop 指令的结构化控制流中，只能通过 end 指令来结束和退出当前指令段的执行流程。也正是由于这个原因，对 loop 指令段的类型检查过程仅会在 end 指令执行时发生。比如下面给出的这段代码。

```

loop $continue i32
...
  get_local $var
  br_if $continue
  i32.const 1
...
end

```

;; 当前位于栈容器顶部的元素是一个值为 1 的整数

这里通过 `loop` 指令创建了一个名为“\$continue”的循环控制结构，并且标记了其返回值的类型签名为 `i32`，即当该段结构化控制流执行完毕时需要向栈容器中压入一个 32 位整数作为其返回值。在这个 `loop` 指令段中，我们通过 `br_if` 指令控制循环的迭代过程。当 `$var` 变量的值不为 0 时，`br_if` 指令会直接对当前的循环结构进行指令指针回溯，即堆栈机会重新从循环结构的第一行指令开始执行。当 `$var` 变量的值为 0 时，`br_if` 指令的跳转条件不成立，指令会继续向下执行。“`i32.const 1`”指令会将一个值为 1 的数字压入栈容器中，随后 `end` 指令会结束整个指令段的执行流程，此时 Wasm 虚拟机才会开始对 `loop` 指令段进行类型检查过程。而之前压入栈容器中的数字 1 便作为了整个指令段的返回值，使得指令段能够顺利地通过类型检查，并成功地被浏览器加载和实例化。

2.3.4 类型堆栈的一致性

我们知道，浏览器在对实际的 Wasm 虚拟指令段进行类型检查时，是通过一个类型堆栈容器来跟踪和检查代码的各种数据及返回值类型是否正确的。对于数据操作指令，堆栈机会根据类型栈容器中当前存放的元素类型来验证它们是否能够被正常执行。同样的，借助类型栈容器，堆栈机还可以对结构化控制流进行类型验证。

比如对于下面给出的这段指令代码，我们可以通过类型栈容器来记录每一条指令运行时数据栈容器中的数据类型变化状态。

```

;; 一些未知的指令
...
f32.const 1
;; 定义了一个结构化的指令段
block $exit
  i64.const 2
  i32.const 3
  ;; 中断当前指令段的执行
  br $exit

```

```
end
```

```
;; 类型栈容器恢复到刚进入结构化指令段时的初始状态
```

上面这段指令代码中的每一条指令运行时类型栈容器的状态变化如图 2-34 所示。这里需要注意，为了方便观察结果，我们将类型栈容器的栈底和栈顶位置进行了颠倒。

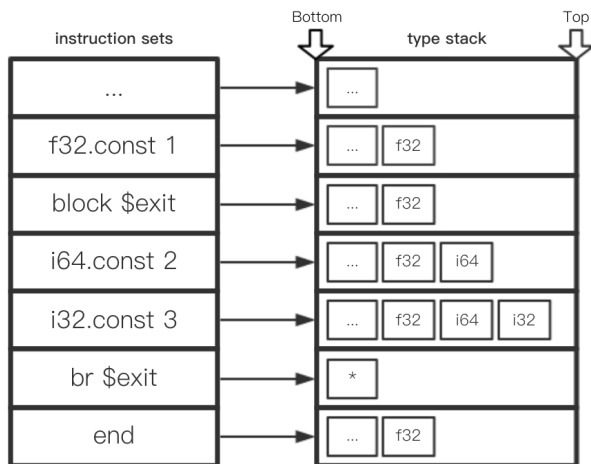


图2-34 上述指令段执行时类型栈容器的变化状态

从图 2-34 中可以很明显地看到，在\$exit 指令段被创建（对应指令“block \$exit”）之前，类型栈容器的状态与\$exit 指令段执行完毕后（对应指令“end”）保持一致，即类型栈容器中记录的数据项个数与其对应签名均一致。当在 if 指令和 block 指令生成的结构化控制流中，通过 br、br_if、br_table 指令来中断当前指令段的执行流程时，中断指令会自动释放栈容器中多余的数据元素，以恢复栈容器中的数据到刚进入该结构化控制流时的初始状态。

需要注意的是，由于 br_if 本身是一个条件中断指令，因此当中断条件不成立时，只能够通过 end 指令来退出指令段的执行流程，这时便需要我们主动将栈容器中的数据恢复至初始状态。比如在下面这段指令代码中，我们尝试通过 br_if 指令来中断一个指令段的执行流程，若中断失败，则会通过 drop 指令将栈容器中的数据恢复到初始状态。

```
;; 一些未知的指令
```

```
...
```

```
f32.const 1
```

```
;; 定义了一个结构化的指令段
```

```
block $exit
```

```
  i64.const 2
```

```
  i32.const 0
```

```
;; 如果当前栈容器的栈顶元素值不为 0，则中断当前指令段的执行流程
br_if $exit
;; 否则手动恢复栈容器的状态
drop
end
;; 类型栈容器恢复到初始状态
```

上面这段指令代码中每一条指令执行时类型栈容器的状态变化如图 2-35 所示。

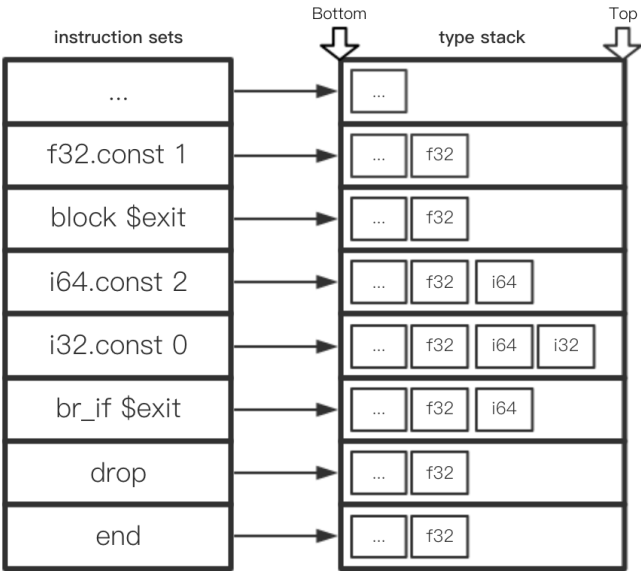


图2-35 上述指令段执行时类型栈容器的状态变化

从图 2-35 中可以看到，`br_if` 指令在进行条件判断时会直接消耗栈容器顶部的一个元素。但是由于中断条件不成立（栈顶元素为 0），且此时栈容器中数据元素的个数仍然多于栈容器的初始数据个数，我们无法直接通过 `end` 指令来结束指令段的执行流程。因此，在这里便需要手动通过 `drop` 指令来移除当前栈容器顶部的一个元素，以便恢复栈容器中的数据至初始状态。

同样的，对于一个由 `if` 指令创建的指令段，从进入该指令段开始到从指令段中退出，一共有三条不同的执行路径：第一条是在指令段中直接通过任意中断指令来退出；第二条是按照正常的指令执行流程，从 `if` 指令段的 `True` 分支（`if`）退出；第三条则是按照正常的指令执行流程，从 `if` 指令段的 `False` 分支（`else`）退出。因此，为了能够通过类型验证，我们需要保证 `if` 指令段在通过这三条执行路径中的任意一条退出后都能够将栈容器中的数据恢复至初始状态。比如下面这段由 `if` 指令构成的代码便无法正确地通过类型验证。

```
;; 一些未知指令
...
;; 获取本地变量$var 的值并压入栈容器中
get_local $var
;; 根据栈容器栈顶元素的值分别进入不同的分支结构中
if i32
    i32.const 2
else
    f32.const 3
end
```

在这段指令代码中，我们为 if 指令段标记了 i32 这种返回值类型签名。因此，当堆栈机从 if 指令段退出时，指令段需要向当前的数据栈容器中压入一个 32 位整数来作为其返回值。一般来说，对于没有标记类型签名的指令段，只需要确保在退出指令段时，栈容器中的数据能够恢复到刚进入指令段时的初始状态即可。而对于标记了类型签名的指令段，也只需要保证堆栈机在退出指令段时，栈容器中的数据状态是在初始状态的基础上新增了一个对应签名类型的数据元素即可。上述虚拟指令代码段执行时对应的类型栈容器状态变化如图 2-36 所示。

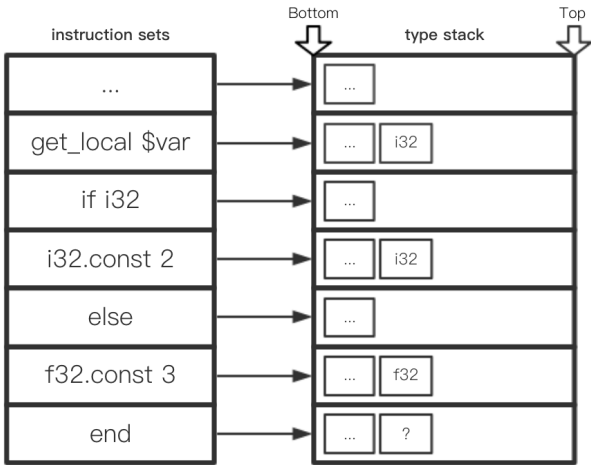


图2-36 上述虚拟指令代码段执行时的类型栈容器状态变化

这里假设变量\$var 的值是一个 i32 类型且值大于 0 的整数，这样便可以让堆栈机进入 if 指令段中执行。从图 2-36 右侧的类型栈容器状态可以看到，堆栈机在刚进入 if 指令段（对应指令“if i32”）时类型栈容器的初始状态是“...”（用于表示“get_local \$var”指令执行前的类型栈容器状态）。因此，当堆栈机从 if 指令段中退出时，该指令段需要在类型栈容器的初始状态基础上再新增一个 i32 类型的元素值。我们知道，每一种结构化控制流的指令段的退出路径都是固

定的，因此只要确保在各条退出路径中都能够使类型栈容器满足特定的数据退出状态即可。在这里我们可以看到，当堆栈机从 if 指令段的 True 分支（if）退出时，栈容器中的数据状态是正确的；而从 False 分支（else）退出时，被推入栈容器的栈顶元素为 f32 类型，类型验证不通过。由于 if 结构化控制流的所有退出分支并没有保持一致的、正确的栈容器退出状态，因此该段指令代码无法被堆栈机正常执行。

2.3.5 不可达代码

顾名思义，所谓“不可达代码”是指一系列无法被堆栈机执行到的指令代码。比如在下面给出的指令代码段中，堆栈机在执行 \$exit 指令段的第一条指令时，便通过无条件跳转指令 br 中断了当前块结构指令段的执行流程，因此位于 br \$exit 和 end 指令之间的所有指令代码都无法再被堆栈机执行到。因此，我们便把这段无法被执行到的代码称为“不可达代码”。

```
block $exit
  ;; 中断当前指令段的执行流程
  br $exit
  ;; 不可达代码段
  i32.const 1
  i32.const 2
  i32.add
  ...
end
```

事实上，由于这部分代码在 Wasm 模块的实际使用过程中是无法被虚拟机执行到的，因此是否对这段代码进行类型验证过程并不会影响模块的实例化和使用效果，哪怕代码中的指令确实不符合类型验证的相关准则。但是，经过 WWG 成员的一系列讨论和决议，最终确定还是需要到 Wasm 模块中的不可达代码进行类型验证，只不过是另外一种特别的方式来进行的。

我们知道，在数据栈容器中可以存放任意数量和类型的元素。因此，对于一段不可达代码来说，我们可以认为栈容器是“多态”的，即可以从栈容器中获取到所需要的任意数量和类型的元素值。所以，验证不可达代码的方式就变成了：是否能够找到一种可以用于成功执行不可达代码的初始栈容器状态。比如在下面的示例代码中，我们直接通过“br \$exit”指令中断了当前代码段的执行流程，因此 i32.add 指令便变成了一个不可达指令。为此，我们需要找到一种初始栈容器的状态可以让 i32.add 指令成功执行。由于 i32.add 指令在执行时需要从栈容器中取出位于栈顶的两个 32 位整型数据，因此在栈顶位置含有至少两个 i32 类型数据元素的初始栈容器状态便可用于成功执行该不可达指令，使得类型验证通过。

```
block $exit
;; 中断代码段的执行流程
br $exit
;; （不可达代码）若可以从多态数据栈中获取两个 i32 类型的操作数，类型验证通过
i32.add
end
```

再比如对于下面这段指令代码，其中的不可达代码段便无法通过类型验证。这是由于 `f32.mul` 指令需要两个 32 位浮点数来作为其指令的操作数，但我们无法找到一种可以用于执行该指令的初始栈容器状态（不论哪一种栈容器状态，位于栈顶的元素都一定是一个 32 位整数 1，而非浮点数类型），因此类型验证失败。

```
block $exit
;; 中断代码段的执行流程
br $exit
;; （不可达代码段）
;; 向栈容器中压入一个值为 1 的数据元素
i32.const 1
;; 操作数类型不匹配，验证失败
f32.mul
end
```

对 Wasm 模块的内部虚拟指令代码进行类型检查，是加载和运行模块前的一个十分重要和必备的操作流程。通过类型检查可以确保模块的定义完好，保证其内部代码的有效性和可用性。特别是伴随着一些模块的运行时检查，我们可以保证模块内的指令代码逻辑不会对规定范围外的内存进行非法访问。而这对于保护浏览器的本地私有资源、防止对栈容器数据进行非法操作有着十分重要的作用。

2.4 二进制编码

在前面的章节中，我们从宏观角度介绍了 Wasm 堆栈式虚拟机的基本原理、模块内部一些常用虚拟指令的用法，以及各类指令段的基本执行规则。但实际上，在 Wasm 模块内部存储的数据信息均是基于二进制格式进行编码的，浏览器在加载和解析模块时也是直接对这些二进制格式的数据流进行处理的。首先，浏览器会根据特定的格式和编码方式将二进制形式的模块数据解码成对应的底层虚拟机指令（OpCode）；然后，浏览器内部用于处理 Wasm 虚拟指令的编译器后端会将这些虚拟机指令转译成平台相关的浏览器汇编代码。最后，这些依赖具体平台的汇编代码便会由 JavaScript 虚拟机来直接执行。接下来，我们将从微观层面入手，来进一步探

索 Wasm 模块内部的“二进制世界”。

2.4.1 字节序——大端模式与小端模式

我们知道，应用程序在运行的过程中，会将所使用到的数据存储于计算机本地的线性内存中。而计算机内存则是由一系列存储单元构成的，每一个存储单元都通过一个唯一的线性地址来标识。在现代的计算机体系架构中，计算机的最小可寻址内存大小为 8 位（bit），即 1 字节（byte）。因此，我们通常将一个存储单元的大小定义为 1 字节。我们通常会将占用多个存储单元的数据称之为“多字节数据”，而这些数据都会被存放到内存中位置连续的存储单元里。

比如在 C/C++ 语言中，一个被声明为 `int` 类型的变量一般会占用 4 字节的内存空间，而这 4 字节的内存空间即对应着内存中实际的 4 个数据存储单元。由于 `int` 类型变量所占用的内存空间超过了 1 字节，因此我们也会称这些数据为多字节数据。

假设我们正在重新设计和制造一台新型计算机，那么在设计多字节数据的线性内存存储方式时，就需要考虑这样一个问题：应该怎样安排多字节数据和内存存储单元的对应方式？由于计算机的线性内存是按照 CPU 的总线地址来进行索引的，因此内存会被划分为低地址段和高地址段两部分，而高地址段的地址值一般会大于低地址段的地址值。同样的，对于一个多字节数据，我们会将该数据二进制形式中的最低位数据和最高位数据分别称为“最低有效位（LSB，Least Significant Bit）”和“最高有效位（MSB，Most Significant Bit）”。比如以在 C/C++ 中声明的 `short` 类型变量值“2597”为例，该值的最高有效位和最低有效位如图 2-37 所示。

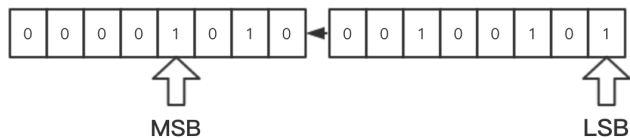


图2-37 `short`类型变量值“2597”在内存中的MSB与LSB

首先，我们声明的 `short` 类型变量在内存中会占用固定的 2 字节空间，即对应两个线性内存的存储单元。在图 2-37 中，每一个小方格都代表内存中的 1 位，8 个小方格构成了一个存储单元，即 1 字节。我们将数字“2597”按照二进制格式展开，便得到了该数字的二进制编码“101000100101”。在该编码中，每一个数字都代表内存中的 1 位。与十进制数字类似，位于该编码最右侧的数字位，我们称之为“最低有效位”，因为该位代表了构成该编码的最低有效权重位，其权重为 1；而位于整个编码最左侧的二进制数字位，我们称之为“最高有效位”，该位代表了构成编码的最高有效权重位，其权重为 2^{n-1} 。其中 n 为该二进制编码的有效位数，这里取

值为 12，因此该编码的最高有效位权重为 2048。

当将多字节数据以二进制形式存储到内存中时，计算机通常会采用两种不同的方式来对应内存位与数字位的存储顺序。第一种方式是将多字节数据的最低有效位存放在线性内存的低地址段，即数据的最低有效位在其最高有效位的前面（低地址段为前），我们称这种数据存储顺序为“小端模式”或“小端序”；第二种方式是将多字节数据的最低有效位存放在线性内存的高地址段，而将最高有效位存放在内存的低地址段，即数据的最低有效位在其最高有效位的后面（高地址段为后），这种数据存储顺序则为“大端模式”或“大端序”。下面我们通过示意图来进一步了解这两种数据存储方式的不同之处。

小端模式（小端序）

如图 2-38 所示，可以看到这里将一个 32 位即 4 字节大小的整数以“小端模式”的数据存储顺序存放到了线性内存中。该整数最高有效位的值为“0A”，这个值被存放到了内存高地址段地址“ $a+3$ ”对应的存储单元中；而该整数最低有效位的值为“0D”，该值被存放到内存低地址段地址“ a ”所对应的存储单元中。如果此时在内存中按照从低地址段到高地址段的顺序依次读取该整数的每一位，则会优先读取到最低有效位，最后才会读取到最高有效位。

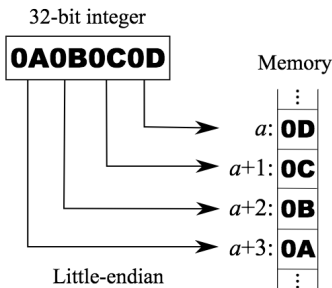


图2-38 小端模式内存结构图（图片来自维基百科）

大端模式（大端序）

如图 2-39 所示，这里将一个同样大小的整数以“大端模式”的数据存储顺序存放到线性内存中。该整数最高有效位的值仍为“0A”，这个值被存放到内存低地址段地址“ a ”对应的存储单元中；而该整数最低有效位的值仍为“0D”，该值被存放到了内存高地址段地址“ $a+3$ ”对应的存储单元中，数据存储顺序正好与“小端模式”相反。如果在内存中按照从低地址段到高地址段的顺序依次读取该整数的每一位，则会优先读取到最高有效位，最后才会读取到最低有效位。需要注意的是，这里提到的“高地址段”与“低地址段”是一对相对的概念，只有在多个地址段相互比较时，才会区分地址段高低的概念。

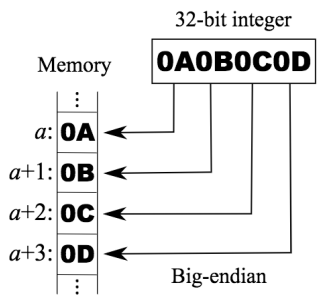


图2-39 大端模式内存结构图（图片来自维基百科）

实际上，大端模式和小端模式两者并没有优劣之分，这两种模式均被广泛地应用在基于不同处理架构的计算机中。比如 PDP-11 等型号的 CPU 处理器会使用小端模式来存储多字节数据，而 Motorola 6800 则会使用大端模式来存储数据。当然，还有一些特殊的 CPU 处理器可以让用户自由地配置多字节数据的存储方式，比如 PowerPC 处理器。从宏观上看，选择使用大端模式还是小端模式意味着 CPU 通过总线进行内存寻址时，是优先读取数据的最高有效位还是最低有效位。换个角度来看，使用大端模式意味着 CPU 会优先读取到数据的最高有效位，而小端模式则会优先读取到数据的最低有效位。

通常我们也会用大端模式与小端模式来表示数据的网络传输顺序。比如在 IPv4/6、TCP 和 UDP 等网络传输协议中规定，数据以大端模式进行传输。即在基于 TCP/IP 协议进行数据传输时，计算机必须优先发送数据的最高有效位。因此，我们也经常称大端模式（大端序）为“网络字节序”。如图 2-40 所示，在 RFC 1700 协议中规定，互联网协议需要使用大端模式来进行网络数据传输（发送与接收数据）。而之所以选择大端序作为网络字节序，原因是作为协议，我们需要选择大端序和小端序两者中的一种，但具体选择哪种实际上并没有太大的区别，大端序只是被最后定下来的那一种而已。

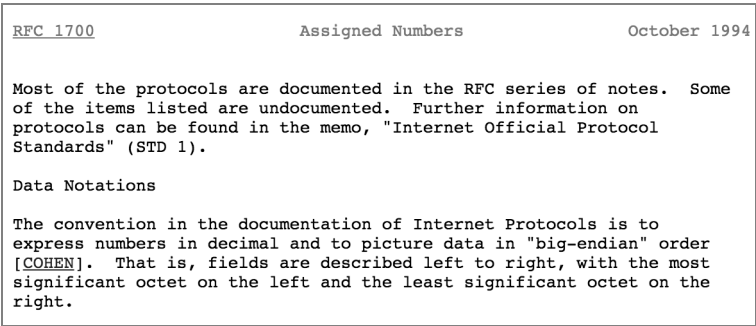


图2-40 RFC 1700协议中关于网络字节序的部分内容

数据优化

接下来，我们再继续深入了解大端模式和小端模式在一些特殊情况下的不同处理方式。小端模式具有这样一个特性，即基于小端模式的 CPU 可以通过不同的数据长度从同一个内存地址处读取到相同的值。比如，一个基于 x86 架构的 CPU 处理器其内存可寻址范围通常为 4Gb，因此这里便假设该处理器的一个内存地址长度应该为 32 位，即 4 字节。现在有一个 int 类型的整数“74”被存放到内存中，如果将该整数转换为十六进制值，则对应的字符串形式为“00 00 00 4A”。其中 4A 为最低有效位所在字节，而按照小端模式的存储方式，该字节将会被存放在内存的低地址段。

假设该位在存储时对应的内存地址为“0x00000001”，此时我们便可以通过该地址来进行数据读取。如果直接将该地址作为首地址，并读取长度为 4 字节的数据，那么得到的数字字面量值为“4A 00 00 00”。由于采用了小端模式，因此还需要对该字面量值进行转换，转换后便可以得到正确的数字字面量值“00 00 00 4A”，即先前存入的数字值 74。接下来仍以该地址作为首地址，并分别读取长度为 3、2、1 字节的数据。我们发现，这些读取到的数据均可以被成功地转换为十六进制数字值“4A”（只是含有占位字节 00）。与“00 00 00 4A”相比尽管取出的数据长度不同，但数据的最高有效位与最低有效位均存在且完全相同，因此对应的十进制数字值也相同。这便是小端模式独有的一个特性。

尽管在实际的应用开发中该特性很少被直接使用，但是在代码优化、编译器实现，以及汇编代码的编写过程中，我们通常会利用该特性来对代码进行一定程度的性能优化。相反，在基于大端模式的 CPU 处理器中，只有完全将用于存放数据的 4 字节存储单元中的内容全部读取出来后，才能够正确地推断出数据的实际值是多少。

计算顺序

小端模式更适合进行加法进位操作。我们知道，在进行数学加法运算时，时常会发生“进位”的情况。比如我们计算表达式“2579 + 74”的最终结果。在计算时，首先会从两个数字的最低位开始进行加法运算。由于是两个十进制数字相加，因此数字中的每一位都需要满足“十进位”的要求，即“满十进一”的规则，这里的“进一”是指向高位进 1。因此，这里两个操作数的个位数字完成相加运算后，需要向十位进 1。同样的，两个操作数的十位数字也需要进行加法运算，但是除两个十位数字本身相加以外，还需要再加上从个位数传来的，加法运算的进位数，即数字 1。操作数后面的所有位数均依此类推。

由于小端模式将数据的最低有效位放置在了内存的低地址段，因此在进行诸如加法的数学运算时，只需要维护一个单调递增的内存数据操作指针，即可完成两个操作数从各自最低有效

位到最高有效位的进位和相加操作。而在大端模式中，首先需要根据数据的基本类型，以及占用的存储单元数，将内存数据指针指向数据最低有效位的字节后，才能开始进行加法运算，并在运算过程中逐渐向内存低地址方向移动数据操作指针。这相对于小端模式来说，可能会带来额外的运算资源开销。

另外，与进行加法运算类似，除法运算的过程需要从数据的最高有效位开始。因此，将数据最高有效位放置在内存低地址段的大端模式会更适合进行除法运算。但实际上，现代的高性能 CPU 处理器都会在其内部内置各种专门用于进行基本数学运算的相关寄存器来加快运算流程，并且处理器会将分散在不同存储单元的数据一次性读出，直接将其存放到寄存器中后再进行运算操作。而 CPU 具体是使用大端模式还是小端模式，基本上不会影响计算效率。

至此，我们介绍完了大端模式和小端模式的相关内容。回过头来看，WebAssembly 在其 MVP 标准中规定，使用小端模式来作为模块文件内容及运行时数据等二进制内容的基本存储方式。但对于一些用于构成模块基本功能的关键字及虚拟指令来说，Wasm 则采用了一种基于小端模式的编码算法对它们进行了可变长编码，这便是下面将要介绍的 LEB-128 编码。

2.4.2 基于 LEB-128 的整数编码

这里要介绍的 LEB-128（Little Endian Base128）是一种用于整数的、基于小端模式的可变长编码。所谓“可变长编码”是指待编码的源数据在经过编码算法后得到的编码结果长度是不固定的。比如我们常用的 UTF-8 编码便是一种可变长编码，其编码结果的长度会随着源数据的不同而发生改变。通过使用可变长编码，可以对源数据进行无损数据压缩，并且被压缩后的数据也可以随时被再次解压缩回源数据。通过对源数据进行合理的编码压缩，可以在一定程度上保证 Wasm 模块的体积大小处在最优的情况下。

通常来说，我们会将 LEB-128 编码分为两种类别。第一种类别为“Unsigned LEB-128”，其主要用于编码无符号整数，并且只能用于编码正整数；第二种类别为“Signed LEB-128”，主要用于编码有符号整数，即该整数对应原码中的最高有效位为符号位。下面通过两个实例来了解这两种编码方式的具体编码流程。

Unsigned LEB-128

这里我们将通过该编码算法来编码一个十进制无符号整数“512384”。

第一步：首先将该十进制数转换为二进制数，转换后得到的结果如下。

```
1111101000110000000
```

第二步：将该二进制数用“0”进行填充，直至其总位数达到最近的一个7的倍数。需要注意的是，只能从该二进制数的最高有效位左侧进行填充，这样才不会改变该数字的值。我们在该二进制数的最左侧填充了两个“0”使其位数正好为7的3倍，即21位。填充后的结果如下。

```
0011111010001100000000
```

第三步：将该二进制数以每7位二进制位为一组，按顺序分成若干组。分组后的结果如下，每组以空格为分隔符进行分隔。

```
0011111 0100011 0000000
```

第四步：在除最高有效位所在分组以外的其他分组的最高位左侧再填充一个值为“1”的二进制比特位，而最高有效位所在的分组则填充值为“0”的二进制比特位。填充后的结果如下。

```
00011111 10100011 10000000
```

第五步：将上述填充好的数据分组依次转换成十六进制数，转换后得到的结果如下。

```
0x1f 0xa3 0x80
```

第六步：由于LEB-128是一种基于小端模式的数据压缩编码算法，编码后的数据需要按照小端序的规则进行存储。因此，这里还需要对编码结果中各个字节的存储顺序进行调整，调整后的数据在内存中的存储状态如图2-41所示。

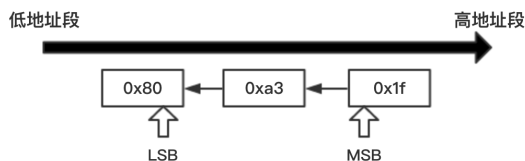


图2-41 数字值在经过“Unsigned LEB-128”编码后的数据存储状态

可以看到经过编码，该整数被转换成了3字节大小的编码值。

Signed LEB-128

这里我们将通过该编码算法来编码一个十进制的有符号整数“-512384”。

第一步：首先将该十进制数转换为二进制数。由于负数的原码、反码和补码三者并不相同，因此需要对其原码进行转换。转换后得到的反码和补码形式如下（这里没有携带符号位）。

○ 原码：1111101000110000000

○ 反码：0000010111001111111

○ 补码：0000010111010000000

第二步：对从上一步操作中获得的二进制数补码进行数据填充，以使其长度能够达到特定的编码要求。这里对有符号整数进行的数据填充操作又被称为“符号扩展”。所谓符号扩展是指通过对某个二进制数填充指定的二进制位来增加其总位数，并保证符号和数值不会被改变。为此，我们需要对负整数使用数字“1”来进行填充，而对正整数还是使用数字“0”来进行填充。对该补码进行符号扩展直至其总位数达到最近的一个 7 的倍数。填充后的结果如下。

1100000101110100000000

第三步：将从上一步中获得的二进制数以每 7 位二进制位为一组，按顺序分成若干组。分组后的结果如下，每组以空格为分隔符进行分隔。

1100000 1011101 0000000

第四步：在除最高有效位所在分组以外的其他分组的最高位左侧再填充一个值为“1”的二进制比特位，而最高有效位所在分组则填充值为“0”的二进制比特位。填充后的结果如下。

01100000 11011101 10000000

第五步：将上述填充好的数据分组依次转换成十六进制数。转换后得到的结果如下。

0x60 0xdd 0x80

第六步：对编码结果中各个字节的存储顺序进行调整，基于小端模式调整后的数据在内存中的存储状态如图 2-42 所示。

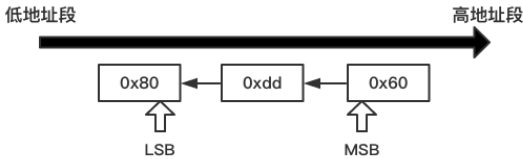


图2-42 数字值在经过“Signed LEB-128”编码后的数据存储状态

综上所述，借助 LEB-128 编码，可以将大小不同的整数编码成不同长度的编码值。而一般来说，随着被编码整数的绝对值不断增大，经过编码处理后生成的编码结果其长度也会相应地增加。

2.4.3 基于 IEEE-754—2008 的浮点数编码

上一节介绍的 LEB-128 编码，主要用于在 Wasm 模块中对部分整数类型的数据值进行编码。

而对于浮点数类型，WebAssembly 在其 MVP 标准中则直接采用了适合表示计算机浮点数的行业标准编码方式，即 IEEE-754—2008 标准。

事实上，这里提到的 IEEE-754—2008 标准，是由其前身老版本的 IEEE-754—1985 标准于 2008 年经过部分修订后重新发布的行业标准。在该标准中规定了浮点数在计算机中的基本存储方式、计算细节，以及一些相关的运算操作等围绕浮点数的标准实现规范。在 IEEE-754—1985 标准的基础上，IEEE-754—2008 标准又增加了诸如 16 位和 128 位二进制类型、3 位小数类型等新的浮点数类型和新的运算操作方法，并同时为标准中的一些术语的话术进行了语法和语义上的修正与增强。由于 IEEE-754—2008 标准并没有对 IEEE-754—1985 标准中浮点数类型的基本存储与展示方式进行修改，因此这里以 IEEE-754—1985 标准为例来进行讲解。

在 IEEE-754—1985 标准中规定，浮点数由三个不同的字段组成，即符号位、指数位和小数位。这里以一个 32 位浮点数“-2579.0451”为例，来介绍浮点数在计算机内存中的具体存放形式。首先，第一个符号位会占用最高有效位，用于标识该浮点数的正负取向，符号位值为“0”代表正数，值为“1”则代表负数。因此这里需要将该位置“1”，如图 2-43 所示。

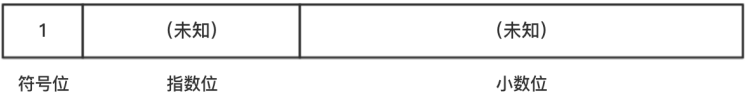


图2-43 使用IEEE-754—1985标准表示浮点数结构

紧接着符号位的是长度为 8 位的指数位，该位用来记录当以科学计数法形式表示一个浮点数时，这个浮点数的幂次值。指数位采用了一种名为“移码”的值计算方式，当我们计算该位的实际值时，会将这 8 位二进制数的十进制值与指数位偏移量“127”进行比较。比如，当指数位的十进制值为 129 时，其浮点数幂次值为 2（129 大于 127）。同理，当指数位的十进制值为 126 时，其浮点数幂次值则为-1（126 小于 127）。

我们首先将该浮点数以科学计数法的形式来表示。但要注意的是，这里需要将该浮点数表示成以 2 为底数的二进制科学计数法形式。经过转换后的浮点数表示如下。

```
换算前: -1 * 2579.0451
换算后: -1 * 101000010011.0000101110001011101011000111000100001100101100101001011
```

可以看到，二进制浮点数与十进制浮点数的计算方式其实是相同的，只不过在计算进位和借位时是采用“2”作为底数的。下面再将这串二进制浮点数整理成以科学计数法表示的幂次形式。整理后的结果如下。

$$-1 * 1.010000100110000101110001011101011000111000100001100101100101001011 * 2^{11}$$

由于该二进制浮点数的小数点向左移动了 11 位，因此科学计数法中底数“2”的幂次记为 11。我们直接在该幂次值的基础上加上 32 位浮点数的指数位偏移量“127”，即可得到该浮点数的指数位的实际存储值，即十进制数“138”，其对应的二进制值为“1001010”，如图 2-44 所示。

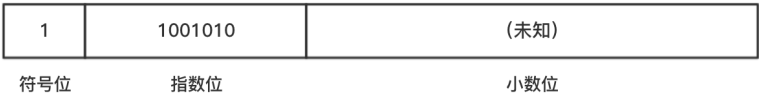


图2-44 该浮点数的“符号位”与“指数位”字段值

接下来再看浮点数的“小数位”，该位主要用于存放浮点数对应二进制科学计数法形式下的小数部分。可以看到，这里该浮点数在对应的科学计数法展开形式下其小数部分长度已经超过了小数位所能够容纳的最多 23 位。因此在实际存放时，计算机会将该二进制浮点数的小数部分进行截断，只将距离小数点最近的 23 位小数存放到内存中，如图 2-45 所示。

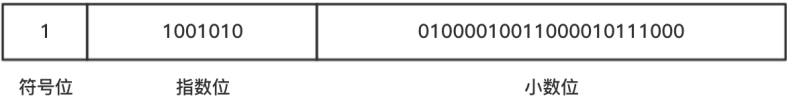


图2-45 使用IEEE-754—1985标准表示浮点数的三个完整字段值

与 32 位浮点数在计算机中的实际物理存储方式一样，64 位浮点数同样由符号位、指数位和小数位三部分组成，只不过其指数位的有效长度为 11 个二进制位，小数位的有效长度为 52 个二进制位，因此可以存储值更大、精度更高的浮点数。

小数位溢出

在日常开发工作中，我们肯定或多或少都听到过“浮点数精度不准”的相关话题。计算机在表示小数时的精度不准是由于在按照十进制数的进位与借位方式，将一个十进制浮点数的小数部分展开成二进制形式时，会发生无法完全展开的问题。比如将浮点数“0.1”的小数部分展开成二进制形式，过程如下。

$$0.1 = 1 * 2^{-4} + 1 * 2^{-5} + 1 * 2^{-8} + 1 * 2^{-9} + 1 * 2^{-12} + 1 * 2^{-13} \dots$$

可以看到，这里我们无法将浮点数“0.1”的小数部分完全展开成对应的二进制形式，而这就导致计算机在实际存储该值时一定会造成精度的损失。由于小数位的长度有限，计算机会将该浮点数小数位多余的部分进行截断处理，这便造成了原始值的精度损失。但也并不是所有计算机都会选择这种方式，在某些编程语言或虚拟机环境中，底层虚拟机可能会采用一种名为“零舍入一”的方式来对小数位中不断重复迭代的小数部分进行处理，而经过处理后的小数位可能

由于进位的原因导致其实际存储值大于原始值。

最大安全数

所谓“最大安全数”是指在该数字值范围以内的所有整数均可以找到与其完全对应的二进制表示形式。这个概念经常在 JavaScript 语言中被提到，这是由于 JavaScript 标准使用 64 位浮点数来表示所有在代码中出现的数字值，而使用浮点数来表示整数便会出现“最大安全数”的问题。我们知道，一个浮点正整数（用浮点数表示的正整数）的大小由该浮点数的指数位与小数位共同来决定。但实际上，指数位却更偏向于确定一个浮点数的颗粒化程度。比如对于一个 32 位长度的浮点正整数，其小数位全部用于描述该正整数的整数值部分，因此其指数位的取值只能大于或等于小数位长度（若小于小数位长度，则会出现小数）。而当指数位减去指数位偏移量后的取值大于其小数位的最大长度时，这个用于构成整数值部分的小数位其最低位的权值便不再是 2^0 ，而变成了 2^1 ，即十进制数值 2。接下来，我们通过下面给出的例子来进一步了解最大安全数的具体含义。

符号位：0

指数位：150 (+23)

小数位：1111 1111 1111 1111 1111 111

这里给出了一个 32 位浮点正整数的各个浮点数组成字段的详细取值情况。此时在该整数值的基础上进行值自增操作，即“加 1”。操作结束后该浮点正整数各个字段的取值情况如下。

符号位：0

指数位：151 (+24)

小数位：1000 0000 0000 0000 0000 000 (0)

可以看到，由于二进制数进位的原因导致小数位的长度发生了变化，其长度从 23 位变成了 24 位。但是由于 32 位浮点数的小数位长度最大为 23 位，因此多余的低位二进制位（位于括号中的）将会被计算机直接截取掉。接下来对该浮点数再进行一次值自增操作，操作结束后各个字段的结果如下。

符号位：0

指数位：151 (+24)

小数位：1000 0000 0000 0000 0000 000 (1)

如果仔细观察，你会发现此时该浮点数在计算机中的具体存放形式并没有发生变化，即各个字段的值均没有发生改变。原因同上。而在进行自增操作时增加的数值“1”，其实是被直接加到了被截取的二进制位上。因此，当计算机将该小数位截断后，其实际存储的小数位值便与之前保持一致了（即该值无法被精确表示）。而这便是“最大安全数”存在的意义。

通常来说，32 位浮点数的最大安全数为 $2^{24}-1$ ，而 64 位浮点数的最大安全数为 $2^{53}-1$ 。需要注意的是，由于可以仅通过浮点数符号位的值来判断该浮点数的正负大小，因此所谓最大安全数实际上是指一个浮点整数绝对值的最大安全取值。

2.4.4 基于 UTF-8 的字符串编码

与各类传统的 Web 应用一样，WebAssembly 同样也选择使用 UTF-8 编码方式来编码各类字符串值。与 LEB-128 编码类似，UTF-8 也是一种可变长编码，即随着被编码内容的改变，编码结果的长度也会发生变化。

UTF-8 的编码过程是基于 Unicode 字符集进行的。在 Unicode 字符集中，每一个字符都有一个与之对应的“码位”，即一个用于标识该字符的序列号码。比如对于汉字“于”，它在 Unicode 字符集中的码位值为“20110”，如果将其换算成十六进制值则为“4e8e”，因此汉字“于”对应的 Unicode 码位值为“U+4E8E”。Unicode 字符集仅规定了各类字符对应的二进制代码，但却没有指出计算机应该将这些二进制代码按照怎样的格式进行存储。而 UTF-8 正是用于规定 Unicode 字符集在计算机中存储方式的众多编码之一。

顾名思义，UTF-8 是一种采用“每 8 位对应一个编码单位”的可变长编码，它会将一个 Unicode 字符集中的码位编码为 1~4 字节对应不同长度的二进制串。这里仍然以汉字“于”为例来说明对 Unicode 字符集的 UTF-8 编码过程。

如图 2-46 所示，这里给出的是不同码位值 Unicode 字符对应的 UTF-8 编码方式。比如，汉字“于”的十六进制码位值为“U+4E8E”，即对应于图 2-46 中第三行的码位编码范围（U+0800~U+FFFF）。整个编码过程如下。

Number of bytes	Bits for code point	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
1	7	U+0000	U+007F	0xxxxxxx			
2	11	U+0080	U+07FF	110xxxxx	10xxxxxx		
3	16	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	21	U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

图2-46 UTF-8编码结构（图片来自维基百科）

- 第一步：先将该码位值展开成二进制串形式，展开后的结果如下。
- 0100111010001110
- 第二步：按照图 2-46 中第三行的规则对该二进制串进行分段。可以看到，在经过 UTF-8

编码后生成的编码结果为一个 3 字节长度的二进制数。该二进制数的首字节表示形式为“1110xxxx”，这里的“x”需要使用上面已经转换好的二进制串来按位进行依次替换。首先按照在每一字节中需要替换的二进制位数量，对该码位值对应的二进制串进行分割。分割后的二进制串结果如下。

```
0100 111010 001110
```

第三步：直接将该二进制串按照分割好的分组，依次对编码结果中每一字节里的“x”符号对应位进行替换。替换后便可以得到汉字“于”经过 UTF-8 编码后的最终结果。

二进制形式：11100100 10111010 10001110

十六进制形式：e4 ba 8e

这便是汉字“于”经过 UTF-8 编码后在计算机中的实际存储值。

2.4.5 模块数据类型

根据 WebAssembly 的 MVP 标准，我们可以将所有用于构成 Wasm 模块的二进制数据分为三种类型，它们分别是：表示无符号整数的 `uintN` 类型、表示可变长无符号整数的 `varuintN` 类型和表示可变长有符号整数的 `varintN` 类型。

`uintN`

该类型表示一个占用 N 个比特位 (bit) 的无符号整数。该整数将以小端模式进行存储，并且占用 $N/8$ 个存储单元（即字节）。 N 的取值可以为 8、16 或 32。

`varuintN`

该类型表示一个基于 LEB-128 编码的可变长无符号整数， N 的取值可以为 1、7 或 32。当对应 N 的不同取值时，该类型的可取值范围为 $[0, 2^N - 1]$ 。比如当 N 的取值为 7 时，`varuint7` 类型的可取值范围为 $[0, 127]$ ，即从 0 到 127（包括两端的值）。需要注意的是，当使用较大范围的数据类型（比如 `varuint32` 类型）来表示较小的数字值时，在 LEB-128 的编码值中可能会含有占位字节 `0x08`。比如对于一个十进制值为 12 的 `varuint32` 类型值，它在 Wasm 模块中可以被表示为如下几种形式。

二进制形式：1100

LEB-128 编码：0x0c

二进制形式：00001100

LEB-128 编码：0x8c 0x00

拟指令指定一个全局唯一的二进制形式的编码值。这些二进制形式的编码值会按照特定的规则被整合到 Wasm 模块的二进制文件中，当浏览器解析一个 Wasm 模块时，Wasm 抽象虚拟机便可以根据这些编码值选择执行不同的虚拟指令操作。我们将这些虚拟指令对应的二进制形式的指令编码称为“OpCode”（本书中，在某些情况下用来直接表示指令本身）。

在最新的 MVP 标准中，WebAssembly 将二进制 Wasm 模块可用的虚拟指令分为如下几种类型。每一种类型的虚拟指令都对应着一系列类似的虚拟机操作。同时，每一种虚拟指令需要的操作数也不尽相同。

结构化控制指令

结构化控制指令主要用于控制模块核心功能中数据的流动方向，以及程序的具体执行逻辑。通过这些指令，我们可以在底层虚拟机上模拟出高级编程语言中的条件语句、选择语句及循环语句等流程控制语句。但是这些虚拟指令的功能更为底层，基于堆栈机实现的结构化控制流使得指令的格式编排与嵌套执行也需要遵循严格的要求。下面给出一段结构化控制指令的示例代码，这些代码均是基于真实的 Wasm 虚拟指令编写的。

```
;; 定义$main代码段
block $main
  ...
  br $main    ;; 中断$exit 指令段
  nop         ;; 不会被执行到的指令代码
end
```

参数操作指令

参数操作指令主要用于对当前数据栈容器中的实参数据进行操作，并且该类型下的指令在执行时并不需要显式指明操作数的具体类型。由于 Wasm 抽象虚拟机是基于堆栈机模型实现的，因此从宏观上看，它会使用一个类似栈结构的容器来保存各种指令操作数、子程序调用的实参，以及一些需要临时存放的中间数据。之前我们介绍过，通过 `br_if` 指令来中断一个指令段的执行流程可能会导致栈容器中的数据状态与其初始状态不一致，这时便需要通过 `drop` 指令来手动地将栈容器中多余的数据元素移除。这里用到的 `drop` 便是一种参数操作指令。下面给出一段参数操作指令的示例代码。

```
;; 一些未知的指令
...
f32.const 1
;; 定义了一个结构化的代码段
block $exit
```

```

i64.const 2
i32.const 0
;; 如果当前栈容器的栈顶元素值不为 0，则中断当前代码段的执行
br_if $exit
;; 手动恢复栈容器的状态
drop
end
;; 将类型栈容器恢复到初始状态

```

内存操作指令

内存操作指令是指一些专门用于操作模块线性内存的虚拟指令。我们之前介绍过，Wasm 模块会在本地维护一个用于和 JavaScript 环境进行数据交互的共享线性内存段。如果将实例化的 Wasm 模块理解为一个运行在浏览器中的应用程序，那么便可以将这个共享线性内存段理解为该应用程序对应 PM 段中的“堆内存段”。通过 load、store 等指令便可以对线性内存段中的数据进行读写操作，这些指令有着众多的“重载”类型，它们可以分别对长度为 1 位、2 位、4 位和 8 位的数据进行存储和写入操作，甚至对这些数据进行符号/零值扩展和截断。下面给出一段内存操作指令的示例代码。

```

;; 定义值分别为 0 和 42 且类型为 32 位整数的常量
i32.const 0
i32.const 42
;; 将值为 42 的 32 位整型数据存储到线性内存索引位置为“0”的存储单元中
i32.store ;; 按 4 字节大小存储数据
...
;; 定义值为 11 且类型为 32 位整数的常量
i32.const 11
;; 从模块线性内存中偏移位置为“11”的存储单元里读取数据并将其存放到数据栈容器中
i64.load32_s ;; 读取 4 字节数据，并将符号扩展至 i64 类型

```

变量操作指令

变量操作指令主要用于定义或访问一个全局变量或子程序中的本地变量。每一个 Wasm 模块都有一个用于存储全局变量的容器向量，在容器向量中记录了所有从模块外部导入或直接在模块内部定义的全局变量。容器会按照单调递增的顺序依次为每一个全局变量分配唯一的索引值，而整个容器会被存放在一段与模块线性内存不相交的独立内存中。

在这个容器向量中存放着每一个全局变量对应的具体值，模块可以借助相应的虚拟指令并通过传递索引值的方式来查找并返回这些全局变量的具体值。全局变量在初始化时会根据其类型被赋予特定的初始值，同时还需要标记出该全局变量的可变类型，即变量的值在后续的程序

运行过程中是否可以发生改变。

本地变量则是指定义在子程序（子函数）中的变量。这些变量只能够在子程序的函数体中进行调用，并且它们的值都是可以随着程序的运行而随时改变的。需要注意的是，在定义子程序时，一些被声明的形式参数也会在真正调用函数时被当作本地变量进行处理。下面给出一段变量操作指令的示例代码。

```
;; 获取本地变量$0 与$1（索引）的值，并压入数据栈容器中
get_local $0
get_local $1
;; 计算位于栈顶的两个元素的数字和，并将计算结果压入栈顶
i32.add
;; 弹出栈顶元素并赋值给全局变量$i
set_global $i
```

数字指令

顾名思义，数字指令主要用于对各种不同类型的数字值进行基础的数学运算和基于栈容器的处理操作。我们知道，计算机的硬件结构决定了它只能处理二进制类型的数据。在一个应用程序内部，计算机会将所有程序使用到的数据通过相应的编码转换成对应二进制形式的数据。同时，对于上层数据的各种处理操作，也都会在底层被转换成直接对二进制形式数据的各种数学运算处理。因此，大部分现代 CPU 处理器为了能够最大程度地提升数据的计算性能，都会将一些常用的数学运算操作直接通过相应的底层硬件结构来实现。同样的，Wasm 标准也选用了能够与底层硬件紧密结合的数学操作来作为抽象虚拟机使用的数字虚拟指令集。我们可以将标准中规定的数字指令分为以下几种类型。

常量指令

常量指令用于直接返回一个静态的数字常量，该常量对应的数字值会被直接压入数据栈容器的顶部。相关的虚拟指令有 `i32.const` 指令等。

一元运算指令

一元运算指令会使用一个操作数并返回一个具有相同数据类型的结果值。相关的虚拟指令有用于计算操作数绝对值的 `i32.abs` 指令等。

二元运算指令

二元运算指令会使用两个操作数并返回一个具有相同数据类型的结果值。相关的虚拟指令

有助于计算两个操作数之和的 `i32.add` 指令等。

测试指令

测试指令会使用一个操作数，并根据该操作数的数字值返回不同布尔值对应的整数。相关的虚拟指令有助于判断操作数的数字值是否为 0 的 `i32.eqz` 指令等。

比较指令

比较指令用于判断两个操作数之间是否满足特定的对应关系，并根据该结果返回不同布尔值对应的整数。相关的虚拟指令有助于判断两个操作数的数字值是否相等的 `i32.eq` 指令等。

转换指令

转换指令会使用一个操作数，并返回与该操作数类型不同的值作为结果。相关的虚拟指令有助于将一个 64 位浮点数转换为 32 位无符号整数的 `i32.trunc_u` 指令等。

下面给出一段数字操作指令的示例代码。

```
;; 定义两个数字常量并将其值压入栈容器中
i32.const 1
i32.const 2
;; 计算上面两个数字常量的和，并将计算结果压入栈容器中
i32.add
```

表达式指令集

表达式指令集并不是指单一的某一类指令，而是指由单一指令所组成的一个指令集合，即对应的一个指令段。比如一个子程序的函数体便是一个表达式指令集。通常来说，一个表达式指令集对应的指令段应该由 `end` 指令来标记其结尾。如下指令代码便是一个由众多单一指令组成的表达式指令集。

```
block $main
;; 一些指令
i32.const 1
i32.const 2
i32.mul
end
```

在当前已经发布的 MVP 标准中，由于 Wasm 虚拟机可以使用的虚拟指令数量小于 256 个，因此这些虚拟指令将会直接按照其 `OpCode` 形式来进行二进制编码，并且 `OpCode` 的编码大小

不会超过 1 字节。但随着 Post-MVP 标准的迭代与发布，伴随新特性而来的则是更多的 Wasm 虚拟指令，因此现有的单字节虚拟指令集其存储方式可能会发生改变。例如，可能会选择一个或多个单字节值来作为多字节指令的表示性前缀。（上述虚拟指令对应的具体 OpCode 值可以参考 WebAssembly 官方技术文档）

2.4.7 类型构造符

在之前介绍的几种类型的虚拟指令中，可以看到大部分指令的字面表示都是由一个点分隔符“.”外加分隔符两边的指令说明符组成的。Wasm 虚拟指令的基本组成结构如图 2-47 所示。

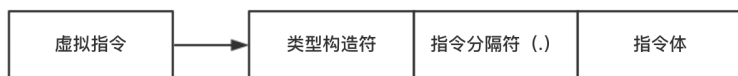


图2-47 Wasm虚拟指令的基本组成结构

每一个虚拟指令都是由特定的三个字段组成的：一个用于说明指令操作数或返回值类型的“类型构造符”、一个具有说明符分隔功能的“指令分隔符”，以及一个用于描述指令本身功能的“指令体”。在 Wasm 标准中规定，虚拟指令的类型构造符一共对应着四种数字类型。

- i32：表示一个 32 位的整数类型。
- i64：表示一个 64 位的整数类型。
- f32：表示一个 32 位的浮点数类型。
- f64：表示一个 64 位的浮点数类型。

另外，指令体字段描述了虚拟指令的具体功能，以及可能的返回值类型。比如以 `i64.load8_u` 指令为例，类型构造符“`i64`”表示该指令在执行后会返回一个 `i64`（64 位整型）类型的数字值作为结果。指令体“`load8_u`”则表示该指令的具体功能是：以 8 位无符号浮点数的格式来读取线性内存中某个特定偏移位置上的数据。通过类型构造符与指令体两个部分的信息，我们便可得到对该指令功能的完整精确描述，即：该指令用于将线性内存段中某个特定索引位置的数据从 8 位无符号整数类型扩展为 64 位无符号整数类型，并返回扩展后的数字值。

综上所述，对于某个虚拟指令的详细功能描述，需要结合该指令的类型构造符和指令体的信息才能确定。尽管每一个虚拟指令的具体功能不尽相同，但是它们的基本描述结构是一样的。

除上述四种可以被用在虚拟指令中的类型构造符之外，在 Wasm 标准中还规定了另外三种

类型构造符，它们在模块特定的二进制代码段组成中有着特殊的标识作用。

- **anyfunc**: 表示带有任意函数签名的函数类型；
- **func**: 表示一个 WebAssembly 的函数定义类型；
- (伪类型): 主要用于表示一个空的逻辑块类型。该类型并没有对应的可用于描述的类型构造符，只有可以应用在模块二进制数据中的 OpCode 代码。

2.5 模块

本节我们将介绍一个标准 Wasm 模块的内部结构，以及构成模块各部分内容的细节信息。我们将 WebAssembly 体系中的一个可以被随意分发、加载及实例化的独立代码单元称为 Wasm 模块。实际上，模块本身就是一个包含了 Wasm 虚拟指令的二进制代码文件，我们可以在浏览器中对它进行实例化并调用其中暴露出的方法。不仅如此，当模块处在实例化过程中时，我们还可以通过从外部环境向模块导入外部数据的方式来控制其最终的实例化状态。

2.5.1 段

我们之前提到过，一个标准的 Wasm 模块是由一系列具有特定功能的“段”结构组成的。在这些段结构中分别存放着用于描述模块功能逻辑和属性的各类信息。不仅如此，模块还定义了一系列用于存放静态资源的索引空间，索引空间内的资源可以由模块中定义的各类指令及相关段结构来进行引用。下面首先介绍模块中定义的各类段结构及其详细功能。

import 段

在 import 段结构中，包含有一系列从当前模块外部宿主环境（如浏览器即为 JavaScript 运行时环境）导入模块的数据项。这些数据会在模块实例化时通过外部接口（如 Web 平台即为 `Webassembly.instantiate` 等类似接口）导入模块中，并直接影响着模块的最终实例化状态。在 MVP 标准中规定，可以被导入 Wasm 模块的 import 段分为如下几种类型。

function import

在 function import 导入结构中包含一簇可以被模块调用的函数。模块在其内部可以通过 call 指令来调用这些从外部宿主环境导入模块中的函数。被导入的函数由两部分组成：一部分为用于描述函数属性信息（参数类型及返回值类型等）的函数签名结构；另一部分为通过函数名或

索引值来进行引用的包含有详细指令及执行流程的函数体结构。函数签名主要被用在模块内部，负责对该导入函数的实际调用过程进行相应的参数及返回值类型校验。

global import

在 `global import` 导入结构中包含一系列从外部宿主环境导入模块中的全局变量。模块可以通过 `get_global`、`set_global` 等指令来操作这些全局变量，直接获取变量的值或为它们赋予新值。被导入模块中的全局变量由两部分组成：一部分为变量的具体值；另一部分为变量的可变性描述符，即该变量是否可以在模块中被随时改变。在目前最新的 MVP 标准中规定，所有从外部宿主环境导入模块中的全局变量其值暂时均不可更改。

linear memory import

在 `linear memory import` 导入结构中包含一段可直接用于模块的线性内存对象。每一个线性内存对象都由两部分组成：一部分为用于描述该内存段初始大小及最大可用大小的属性描述符；另一部分为对应于该线性内存对象实际分配的内存空间引用。

一个有效的线性内存对象必须同时符合两个要求，即其实际引用的内存空间大小不能小于在属性描述符中规定的初始内存大小并且不能大于同样在属性描述符中规定的最大可用大小。在 MVP 标准中，每一个实例化的 Wasm 模块都会自动生成一个默认且唯一的共享线性内存段。因此，从外部宿主环境导入模块中的线性内存对象最多也只能有一个。

模块的线性内存主要用于在外部宿主环境和模块对象本身之间创建一个可以进行数据共享的内存空间。一般来说，可以通过优先在宿主环境内手动构造共享内存段对象并通过实例化方法将其传递给模块使用的方式，来完成需要在模块初始化时进行的数据共享过程。

table import

在 `table import` 导入结构中包含一个可用于进行 WebAssembly 模块动态链接的“间接函数调用表”结构。模块可以通过 `call_indirect` 指令来调用存放在表元素段中的函数。

一个有效的表结构需要同时符合两个特定的要求：首先，表结构中存放的元素类型需要与其在属性描述符中定义的类型保持一致；其次，实际引用的表结构对象其可用长度（可存放元素的个数）不能小于在属性描述符中规定的表初始长度，同时也不能大于在属性描述符中规定的表最大长度（可选项，不指定则不限制长度）。

另外需要注意的是，每一个具体的 `import` 段都包含有两个特殊的标识符，其中一个用来标

识该段所在的具体命名空间名；另一个则用来标识段的具体名称。每一个标识符都需要通过 UTF-8 进行有效编码。这里我们从某个标准 Wasm 模块对应的可读文本格式中截取了一段在该模块中定义的 `import` 段内容，这部分虚拟指令的内容如下，可以作为参考。

```
...
(import "env" "DYNAMICTOP_PTR" (global (;0;) i32))
(import "env" "STACKTOP" (global (;1;) i32))
(import "env" "STACK_MAX" (global (;2;) i32))
(import "env" "abort" (func (;0;) (type 5)))
(import "env" "enlargeMemory" (func (;1;) (type 7)))
(import "env" "getTotalMemory" (func (;2;) (type 7)))
...
```

可以看到，在这段 Wasm 可读文本代码中一共定义了从外部宿主环境导入的 6 个 `import` 段，它们均属于同一个名为“`env`”的命名空间，并且有着自己的名称和类型。这里我们以上面第二行虚拟指令代码为例来介绍一个 `import` 段对应的指令组成结构。

这行以“S-表达式”形式表现的虚拟指令表达式声明了模块需要从外部宿主环境导入的一个 `import` 段结构。这个 `import` 段结构被声明在名为“`env`”的命名空间中，其别名“`STACKTOP`”可以直观反映出该段结构的具体功能和作用。接下来的参数告诉我们该 `import` 段结构的具体类型为“`global import`”，即导入模块内部的是一些由外部宿主环境定义的全局变量。而在 `global` 类型的详细定义参数中，“`(;1;)`”参数指明了该全局变量在模块内对应资源索引空间中的具体索引位置，“`i32`”参数指明了该全局变量值的具体类型。对于其他类型的 `import` 段结构，我们也可以通过类似的方式进行结构化分析。

export 段

既然可以从外部宿主环境向模块导入特定类型的数据，那么反过来也可以将这些数据从模块内导出到模块外的宿主环境中。被导出的可用 `export` 段类型与 `import` 段保持一致。

每一个从模块导出的 `export` 段结构都由三个主要部分构成：一个用于标识该导出结构的别名，该别名需要以 UTF-8 格式进行编码，并在所有的 `export` 段结构中保持唯一；一个用于标识该导出结构对应具体类型的说明符；一个表示该类型对应实体在资源索引空间中具体位置的索引值。我们从某个标准的 Wasm 模块可读文本代码中截取了一段在该模块中定义的 `export` 段内容，详情如下。

```
...
(export "_malloc" (func 48))
(export "_getArrayOffset" (func 24))
```

```
(export "getTempRet0" (func 23))
(export "_fflush" (func 46))
(export "runPostSets" (func 68))
(export "_sort" (func 27))
...
```

这里我们以第一行虚拟指令代码为例，来介绍 `export` 段在 Wasm 模块可读文本代码（.wat）中的具体组成方式。

这行代码同样是以“S-表达式”的形式进行表达的。首先，“`_malloc`”作为别名直接反映出该 `export` 段具有的实体功能（即用于内存分配的函数）。接下来的“`(func 48)`”字段表明该导出段的具体类型为一个“函数导出段”，即导出到外部宿主环境中的内容是一个标准的 WebAssembly 函数。字段最后的数字“48”指明该函数对应实体在当前模块函数资源索引空间中的具体索引位置。

这里需要注意的是，所有导出到外部宿主环境中的 `export` 对象的具体功能均是由宿主环境决定的。比如一个从 Wasm 模块中导出的标准 WebAssembly 函数，在以浏览器为主的外部宿主环境中会被直接转换成一个签名对应的 JavaScript 函数。这种从 Wasm 函数到 JavaScript 函数的转换规则便是由宿主环境定义和实现的。类似的，对于其他 Wasm 标准中的数据定义，其在两个环境中的数据转换规则，也会根据宿主平台的不同而发生改变。

start 段

`start` 段为我们提供了一个可以对 Wasm 模块设置初始化钩子函数（Hook）的功能。所谓钩子函数，是指该函数的执行过程与模块的实际生命周期直接挂钩。如果在一个 Wasm 模块中定义了 `start` 段，该模块便会在完成实例化后直接立即调用在 `start` 段中指定的钩子函数。而对于这些可以被应用在 `start` 段中的 Wasm 函数，它们必须满足以下要求。

- 该函数不可以带有任何形式参数，同时不能有任何返回值。
- 该函数必须被定义在模块的全局函数资源索引空间中，函数可以选择从外部宿主环境中导入，并且可以被任意地从模块中导出。
- 在一个 Wasm 模块中只能设置一个 `start` 段。

下面给出的是在某个标准 Wasm 模块中定义的 `start` 段代码。该段虚拟指令代码仍然是以 S-表达式形式表达的 WebAssembly 可读文本代码。

```
(module
  (start 2) ;; 或者 (start $_Z6outputv)
```

```
(import "env" "printf" (func $printf (param i32 i32) (result i32)))
...
(func $_Z6outputv (; 2 ;))
  (local $0 i32)
  (i32.store offset=4
    (i32.const 0))
...
```

可以看到，代码的第二行通过（start 2）这种指令形式设置了该模块在实例化后可以调用的钩子函数。该字段中的数字“2”表示钩子函数在模块函数资源索引空间中的实际索引位置。我们继续沿着代码向下查找便可发现，该索引位置对应函数的别名为“\$_Z6outputv”，因此实际上我们也可以通过（start \$_Z6outputv）这种“引用资源别名”的指令形式来设置模块 start 段的内容。总的来说，在 Wasm 模块中我们可以通过索引值或别名的方式来引用模块各类型资源索引空间中的大部分资源对象。而 start 段则为我们提供了可以为模块绑定实例化回调函数的功能。

global 段

global 段结构用来存放 Wasm 模块在其内部定义的全局变量。每一个全局变量的定义结构都由三部分组成：一部分用于标识该全局变量在模块全局变量资源索引空间中具体位置的索引值或 UTF8 别名；一部分用于标识该全局变量具体数据类型及可变型的标签属性；一部分用于确定全局变量初始值的初始化表达式。同样的，我们还是通过一段在某个标准 Wasm 模块中定义的 global 段指令代码来理解该段结构的具体组成方式，如下所示。

```
...
(global (;5;) (mut i32) (i32.const 1))
(global (;6;) (mut i32) (get_global 0))
(global (;7;) (mut i32) (get_global 1))
(global (;8;) (mut i32) (i32.const 0))
...
```

以上述指令代码段中的第一行指令表达式为例。首先，该指令表达式中的“(;5;)”字段表明全局变量在模块全局变量资源索引空间中的索引位置，这里指索引值为“5”的位置。接下来的“(mut i32)”字段表明该全局变量为一个可变型（Mutable）且值为 i32 类型的变量。所谓的可变型是指全局变量的值可以在模块初始化后被模块内的指令随时改变。第三个字段“(i32.const 1)”为该全局变量的初始化表达式，该表达式将一个 32 位的整数值“1”赋值给该全局变量以作为其初始值。

对于全局变量的初始化表达式，MVP 标准对其有着严格的格式组成要求。标准中规定，一个全局变量的初始化表达式是由一小部分的 WebAssembly 虚拟指令表达式子集构成的。而对于

在当前 MVP 标准中规定的初始化表达式，则只能由如下两类虚拟指令运算符构成。

- `i32.const` / `i64.const` / `f32.const` / `f64.const`
- `get_global`

第一类运算符是在 WebAssembly 标准中定义的常量运算符，通过这类运算符可以直接为全局变量设置初始值；第二类运算符可以间接地将一个 `global` 导入段对应的数字值赋值给全局变量作为其初始值。但这里要注意的是，对应的 `global` 导入段必须被标识为不可变类型。需要注意的是，初始化表达式的概念不止被用在这里对全局变量的值初始化上，模块内其他需要进行值初始化的地方都可能用到。

memory 段

`memory` 段结构主要用于定义模块实例化后的可用线性内存段。在 MVP 标准中规定，每一个标准的 Wasm 模块都可以选择使用一个模块默认生成的，或者是从外部环境导入的线性内存段。每一个线性内存段的 `memory` 段结构都由两个字段组成：第一个是用于指定内存段初始大小的 `initial` 字段；第二个是用于指定内存段最大可扩展大小的 `maximum` 字段。这两个字段的值均会以内存“页”为单位（Wasm32 中为 64KB）直接反映到对应大小的内存容量上。

同样是截取自某个标准 Wasm 模块中的 `memory` 段定义表达式，具体的段结构定义如下。

```
...
(memory (;0;) 1 10)
...
```

该类型段结构主要由三部分构成。“(;0;)”指明该线性内存段在模块线性内存资源索引空间中的具体索引位置。由于在 MVP 标准中规定，每个 Wasm 模块最多只能够拥有一个线性内存段，因此这里在定义 `memory` 段时标记的索引位置“0”表示模块的第 0 个内存段，即仅有的那一个。但为了方便技术实现，在当前 WebAssembly 虚拟机的底层实现逻辑中可能并没有专门构建用于存储资源索引值的索引空间，而是采用判断模块是否含有“`memory`”占位符的方式来标识模块是否拥有线性内存段。

接下来的数字“1”指明了线性内存段的初始容量大小，对应一个内存页即 64KB 大小的内存空间。最后的数字“10”标记了该线性内存段的最大可用内存大小，对应于 10 个 WebAssembly 内存页即 640KB 大小的内存空间。当模块被实例化后，虚拟机会默认为其自动生成一个初始容量为一个内存页大小的线性内存空间。随着 Wasm 应用的不断运行和使用，其可用的线性内存会逐渐减少，此时我们便可以通过 `grow_memory` 指令来增加线性内存的容量。但是可增加到的

最大内存容量不能超过该模块 memory 段中标记的最大可用内存大小。

data 段

在 Wasm 模块中 data 段的主要功能是用于指定模块线性内存段的初始数据内容。若在一个 Wasm 模块中没有定义相应的 data 段，则对应的线性内存段数据会被初始化成值为“0”的字节数据。

data 段的定义结构主要由两部分组成。第一部分为需要初始化的线性内存段偏移量。在这个内存偏移量字段中，我们可以通过 WebAssembly 的常量表达式来指定内存偏移字节数。比如常量表达式“i32.const 16”字段便指定了一个大小为 64 字节的内存偏移量。第二部分为需要初始化的具体内存数据内容。初始化数据的长度并不需要完全匹配整个线性内存段的大小，若初始化数据的长度小于模块在 data 段中指定的线性内存偏移量，则剩余的内存空间将会被自动填充成值为“0”的字节数据。

现阶段在 MVP 标准中规定，一个 Wasm 模块最多只能拥有一个线性内存段，因此所有在 data 段中指定的线性内存段均对应于模块仅有的那一个线性内存段。在定义该段结构时，暂时并不需要显式指出内存段在模块线性内存资源索引空间中的具体索引位置。这里还是以分析实际 Wasm 模块代码为例来了解 data 段的具体组成方式。如下是一个 data 段定义表达式对应的 Wasm 可读文本代码。

```
...  
(data (i32.const 16) "%d\00")  
...
```

首先，表达式中的第一个字段“(i32.const 16)”指出该内存段需要填充的数据偏移量为 16 个 32 位整数的长度，即 64 字节。第二个字段“%d\00”则为默认的模块线性内存段初始化数据。这里由于 WebAssembly 的可读文本代码（.wat）采用了“S-表达式”作为模块内部虚拟指令结构的字面展示方式，而“S-表达式”本身又对其用于展示的这部分可读文本代码内容进行了 ASCII 编码，因此只要将上述代码表达式描述的 data 段结构中用于初始化线性内存段的数据进行相应的 ASCII 解码，便可得到内存中实际存储的二进制内容。

需要注意的是，data 段实际上并不关注模块线性内存段中初始化数据的实际数据类型，所有的数据均以字节为单位进行存储。数据的使用方式由具体的指令代码逻辑来决定。

table 段

在详细介绍 table 段之前，我们先来了解一下 WebAssembly 在其标准中制定的“Table”元

素是一个什么样的概念。我们可以将这里的 `Table` 元素直接翻译为“表”元素。实际上，表元素对象与模块中的共享线性内存段十分相似，只不过在这个表结构中存储的并不是二进制形式的字节数据，而是一些符合特定“表类型”的元素值。整个表结构被划分为多个不同的“小格子”，每一个小格子都有一个唯一的整数索引值，在模块中我们可以通过相应的虚拟指令，并通过该索引值来获取这些小格子中的内容。

如果需要在表结构中存放一些元素值，则这些元素值的数据类型必须严格符合模块在其 `table` 段中指定的数据类型。同样的，我们也可以通过 `WebAssembly.Table.prototype.set` 等方法来从模块的外部宿主环境向模块的表结构中存放数据。在当前 MVP 标准中，唯一可用的表结构元素类型只有 `anyfunc`，即对应标准中的 WebAssembly 函数类型。

表结构除对存入其内部的数据类型有严格的要求之外，表结构本身也是有固定的表空间个数的。在模块的 `table` 段中，明确地指定了该模块表结构的初始表空间个数，以及最大可用表空间个数，即表结构中可用于存储数据的“小格子”数量。在现阶段标准中，我们可以在模块的外部宿主环境中通过上层 JS 接口 `WebAssembly.Table.prototype.grow` 来增加表结构的表空间个数，但增加后的最大表空间个数不能超过模块在 `table` 段中定义的最大可用表空间个数。

在当前情况下（MVP 标准），只能在模块的表结构中存放标准 WebAssembly 函数的引用指针。因此对应用于从表结构中读取数据的接口，我们只能通过 `call_direct` 虚拟指令来访问一个被标记为“`anyfunc`”类型的表结构。访问的最终效果是模块将会直接利用表结构对应索引位置上存放的函数指针来调用其对应的函数体。

相比同样用于数据存储的模块线性内存段来说，在表结构中存放诸如“函数指针”“原始的系统指针”“GC 引用”等不透明值的优势在于，我们不需要将这些抽象类型以具体的二进制数据形式进行存储和使用。这里的不透明值是指对于一些具有抽象类型的数据结构，我们无法直接对这些数据本身的二进制编码进行处理，但是借助模块自身或由模块外部宿主环境提供的一些特定的方法接口，便可以对这些抽象数据进行相应的操作，而不用关心这些接口在二进制编码的层次上对数据做出了哪些改变，只需关注接口的调用结果即可。另外，不同于线性内存段需要通过详细的内存偏移地址来进行数据索引，表结构仅需要使用简单的 32 位整数索引值便可以进行相应的数据索引操作，这使得在数据索引过程中进行的边界检查过程变得更为简单和快速。

可以通如下虚拟指令表达式来定义一个最简单的 `table` 段，这里同样以“S-表达式”形式展示的 Wasm 可读文本代码。

```
(table 1 10 anyfunc)
```

上述代码定义了 Wasm 模块中的一个 table 段。其中第一个字段“1”表示该模块表结构的初始表空间个数为 1；第二个字段“10”指出该模块表结构可以被增加到的最大表空间个数为 10；第三个字段“anyfunc”表明在该表结构中仅可存放类型为标准 WebAssembly 函数引用指针的数据元素。

在当前的 MVP 标准中，由于模块表结构的功能极其有限，可以存储的数据类型和可用的数据操作接口较为单一，因此其主要用来作为不同 Wasm 模块间进行函数动态链接时的函数容器或者基本的函数间接调用表。总结来看，基于当前 MVP 标准的模块表结构，其在功能上存在如下限制。

- 模块表结构对应表空间中的数据内容当前只能通过 WebAssembly 虚拟指令 `call_indirect` 进行间接读取和调用。
- 当前唯一可用的表结构数据类型为 `anyfunc`，即带有任意函数签名的 WebAssembly 子函数对应的引用指针。
- 表结构的可用表空间个数不能直接通过 WebAssembly 相关虚拟指令在模块内部进行调整，而只能够在模块外部的宿主环境中通过相应的上层接口（如 JavaScript）进行调整。

在当前的 MVP 标准中规定，我们只能够在在一个模块中定义或从外部导入一个表结构对象。这个限制将会在未来的标准迭代中逐渐被取消。

elements 段

`elements` 段主要用于为模块内的表结构提供初始化内容。一个没有经过 `elements` 段对其进行内容初始化的表结构，其表空间的内容不可以被直接读；否则，该操作会向模块外部的宿主环境中抛出一个特定的异常。在下面的例子中，给出了几种在模块中定义 `elements` 段的不同方式，这些方式仍然是按照“S-表达式”形式的可读文本代码进行描述的。

```
...
(table $t 10 anyfunc)
(func $f)
(elem 0 (offset (i32.const 0)) $f $f)
(elem $t (i32.const 0) $f $f)
...
```

首先，这段代码的第一行定义了一个大小为“10”个表空间的模块表结构，并且指定该表结构只能存储类型为“anyfunc”的元素值。这里的“\$t”为该表结构的别名。第二行代码定义了一个名为“\$f”的函数，但是并未为该函数指定函数体。接下来的两行代码定义了两个用于

对表结构进行内容初始化的 `elements` 段，虽然它们的具体表达细节并不完全相同，但是其对表结构填充的初始化数据内容却是完全相同的。这也是 `elements` 段的常见使用方式。

第三行代码首先通过索引字段“0”来指定要进行内容初始化的表结构。由于在当前的 MVP 标准中规定，每一个模块只能有一个被导入的或在模块中定义的表结构，因此这里的索引值“0”便直接特指模块内部唯一的表结构。接下来的“`(offset (i32.const 0))`”字段指定 `elements` 段要进行内容初始化的表结构位置偏移量，即进行内容初始化的具体表空间位置。这里的取值“0”表示从表结构的第一个表空间开始进行数据初始化。具体的初始化内容便紧接着被依次排列在该字段的后面，以空格进行分隔，每个元素对应于一个表空间，表空间的具体位置索引值从偏移量处开始依次递增。这里模块需要将两个函数“`$f`”的函数指针存放到表结构“`$t`”的 0 和 1 索引位置对应的表空间中。此时该表结构中对表空间的数据状态如图 2-48 所示。



图2-48 上述表结构中对表空间的数据状态

从图 2-48 中可以看到，当前已经为该表结构中的前两个表空间填充了具体的初始化内容。对于表结构中那些没有进行内容初始化的表空间，仍然不可以对其进行数据访问操作。

function/code 段

`function/code` 段主要用来存储在模块中定义的所有函数类型数据。在一个标准的 Wasm 模块中，我们会将一个定义完好的函数结构分为两个部分来分别进行存储（类似 C/C++ 中的函数声明和函数定义）。第一个部分是由函数的函数名、形式参数和返回值类型构成的函数签名，模块中的每一个函数的函数签名必须保持唯一。第二部分为函数的实际函数体，在函数体中存放着函数实际运行的逻辑指令段。`function/code` 段则分别用于存储一个函数结构对应的这两个部分。其中 `function` 段主要用于存储函数的函数签名部分；`code` 段则主要用于存储函数签名对应的实际函数体部分。

Wasm 模块之所以选择将函数定义分为独立的两个部分进行单独存储，是由于这种方式便于外部宿主环境更高效地进行流式编译。在前面章节中我们介绍过，诸如 `WebAssembly.compileStreaming` 等宿主环境提供的 API 接口可以使我们以底层数据流的方式来编译一个 Wasm 模块。因此，将模块中定义的函数的函数体和函数签名分开，使得宿主环境在以底层数据流方式编译模块时，可以优先获取用于描述函数信息的函数签名，即用于描述模块特征的相关“元信息”。这些元信息可以让虚拟机优先对模块进行类型检查的过程，同时使模块后续的远程数据

加载和本地编译两个过程可以同时进行，而这会大大缩短模块的初始化时间，提升其使用效率。

总的来说，从宏观上看，Wasm 模块的整个二进制数据内容被分割为不同类型的“段”结构，每一种类型的段结构都负责表达模块某一方面的组成信息，而这些段结构之间的相互依赖关系又组成了模块完整的功能和逻辑体系。在这一节中，我们曾多次提到 Wasm 模块的可读文本代码，即基于 S-表达式进行描述的 Wasm 模块代码内容。这里需要注意的是，所谓的可读文本代码是指通过某种语法和语义表达方式，将一个 Wasm 模块的完整内容以人类可以读懂的方式表达出来。但实际上，在 Wasm 模块中存储的只有原始的二进制数据，这些数据一部分是对应某些虚拟指令的 OpCode 代码；一部分是用于标识段结构和模块信息的元数据，但不论是哪种类型的信息，它们都是以二进制形式被存放在模块文件中的。

2.5.2 索引空间

在本章的开头我们提到过，被定义在 Wasm 模块中的索引空间主要用于存储一系列模块中各种“段”结构使用到的静态资源，这些静态资源可以被相应的段结构或虚拟指令运算符进行静态引用。这里需要注意的是，所谓的索引空间是一种抽象的宏观概念，我们可以将索引空间理解为一组“目录”结构（从宏观上看），利用这些目录结构便可以通过具体的“页码值（索引值/别名）”查找到各种资源。但在实际的模块二进制代码中，可能找不到对应实际表现出来的索引空间结构。在模块中我们能够使用到的索引空间主要有如下四种类型，对这些索引空间的详细介绍均基于 WebAssembly 当前最新的 MVP 标准。

Function 索引空间

Function 索引空间主要用来存储所有直接在模块中定义或从外部宿主环境导入模块中的函数定义。在该索引空间中可以存放多个函数定义元素。

Global 索引空间

Global 索引空间主要用来存储所有直接在模块中定义的或从外部宿主环境导入模块中的全局变量定义。在该索引空间中可以存放多个全局变量定义。

Linear Memory 索引空间

Linear Memory 索引空间主要用来存储所有直接在模块中定义的或从外部宿主环境导入模块中的共享线性内存段定义。在该索引空间中仅允许存放一个共享线性内存段定义。

Table 索引空间

Table 索引空间主要用来存储所有直接在模块中定义的或从外部宿主环境导入模块中的表

结构定义。在该索引空间中仅允许存放一个表结构定义。

在这些索引空间中，模块会按照单调递增的顺序，从索引值“0”开始依次为每一个存储在索引空间中的元素分配唯一的索引值，从模块外部导入的元素将会被优先分配索引值。再次强调：从宏观上看，所谓的索引空间只是在 Wasm 模块中存在的一种概念性的数据结构而已。从实际的模块二进制代码中可能找不到任何用于定义索引空间的虚拟指令关键字，只是模块内部的整体逻辑执行方式表现出了“索引空间”这一概念。下一节我们将介绍模块在“微观”二进制数据层面的具体组成形式。

2.5.3 二进制原型结构

本节我们从创建一个简单且完整的 Wasm 应用开始讲起。我们知道，一个最简单的 Wasm 应用由两部分组成：一个标准的 Wasm 模块和一段用于连接模块与 Web 宿主环境（浏览器）的 JavaScript 脚本代码。首先，我们需要通过 WasmFiddle 平台来创建一个标准的 Wasm 模块。下面给出的是该模块的 C/C++源代码。

```
int main() {  
    return 0;  
}  
  
int add (int a, int b) {  
    return a + b;  
}  
  
double add (double a, double b) {  
    return a + b;  
}  
  
int minus (int a, int b) {  
    return a - b;  
}
```

这是一段十分简单的 C++源代码，我们在代码中一共定义了三个函数，分别用于计算两个整数的数学和、两个浮点数的数学和，以及两个整数的数学差值。由于前两个函数的函数名相同，形式参数类型与返回值类型不同，因此我们需要利用 C++语言中的重载特性来编译这段代码。这里需要设置 WasmFiddle 平台的编译参数，让其能够以“-std=C++11”编译器标识即 C++ 11 的语法特性来编译该模块。我们可以直接点击 WasmFiddle 平台右上角的设置按钮，并按照图 2-49 所示的方式设置 Compiler Options 参数。

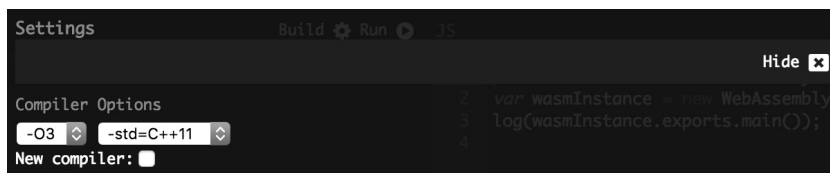


图2-49 在WasmFiddle平台上设置基本的编译器参数

设置好编译器的相关参数后，便可以开始模块的编译过程了。模块编译完成后，我们可以在 WasmFiddle 平台最下方的输出窗口（选择 Text Format 选项）中得到该 Wasm 模块对应的可读文本格式（WAT）代码。可读文本代码以“S-表达式”这种方便人类阅读的语义和语法表达方式，将模块内部的二进制虚拟指令和代码逻辑展示出来。该模块的可读文本代码如下。为了便于书中后续相关内容的展开，我们对这段 WAT 代码进行了部分修改。

```
(module
  (table $0 1 anyfunc)
  (memory $0 1)
  (export "memory" (memory $0))
  (export "main" (func $main))
  (export "_Z3addii" (func $_Z3addii))
  (export "_Z3adddd" (func $_Z3adddd))
  (export "_Z5minusii" (func $_Z5minusii))
  (elem $0 (i32.const 0) $_Z3addii)
  (func $main (; 0 ;) (result i32)
    (i32.const 0)
  )
  (func $_Z3addii (; 1 ;) (param $0 i32) (param $1 i32) (result i32)
    (i32.add
      (get_local $1)
      (get_local $0)
    )
  )
  (func $_Z3adddd (; 2 ;) (param $0 f64) (param $1 f64) (result f64)
    (f64.add
      (get_local $0)
      (get_local $1)
    )
  )
  (func $_Z5minusii (; 3 ;) (param $0 i32) (param $1 i32) (result i32)
    (i32.sub
      (get_local $0)
```

```

    (get_local $1)
  )
)
)

```

最后，我们可以通过如下 JavaScript 代码来连接宿主环境与 Wasm 模块，并调用模块暴露出的一系列方法（将 JavaScript 代码直接输入 WasmFiddle 平台右侧的 JavaScript 脚本区域中，然后点击菜单栏中的“运行”按钮即可编译并运行这个 Wasm 应用）。这里我们回忆一下，从这段 JavaScript 代码中可以看到，从模块中导出的函数名与我们在 C++源代码中定义的函数名并不相同，而这是由于 C++语言本身的 Name Mangling 机制导致的。

```

const wasmModule = new WebAssembly.Module(wasmCode);
const wasmInstance = new WebAssembly.Instance(wasmModule, wasmImports);
const { _Z3adddd, _Z3addii, _Z5minusii } = wasmInstance.exports;
// 调用 WasmFiddle 平台提供的打印函数来输出模块函数的调用结果
log(_Z3adddd(2, 2));

```

至此，一个基于 WasmFiddle 构建的简单 Wasm 应用便创建完成了。现在我们把目光放到 Wasm 模块上。首先，我们通过 WebAssembly 官方为开发者提供的 WABT 工具链中的 wat2wasm 工具，将 WasmFiddle 平台上模块的可读文本代码转换成一个完整的二进制模块文件。接下来，便可以在计算机的本地命令行终端中，通过 hexdump 命令以十六进制数据和对应 ASCII 编码值的形式来查看模块中的二进制内容。这里需要执行的完整命令为“hexdump -C program.wasm”，命令中的参数“program.wasm”为上述 Wasm 二进制模块的文件名，“-C”选项告知该命令我们所需要的数据展示形式（hex+ASCII）。命令执行后输出的信息如图 2-50 所示。

```

→ Desktop hexdump -C program.wasm
00000000  00 61 73 6d 01 00 00 00 01 91 80 80 80 00 03 60  |.asm.....`|
00000010  00 01 7f 60 02 7f 7f 01 7f 60 02 7c 7c 01 7c 03  |...`....|.|.|
00000020  85 80 80 80 00 04 00 01 02 01 04 84 80 80 80 00  |.....|
00000030  01 70 00 00 05 83 80 80 80 00 01 00 01 06 81 80  |.p.....|
00000040  80 80 00 00 07 b4 80 80 80 00 05 06 6d 65 6d 6f  |.....memo|
00000050  72 79 02 00 04 6d 61 69 6e 00 00 08 5f 5a 33 61  |ry...main..._Z3a|
00000060  64 64 69 69 00 01 08 5f 5a 33 61 64 64 64 64 00  |ddii..._Z3adddd.|
00000070  02 0a 5f 5a 35 6d 69 6e 75 73 69 69 00 03 0a ae  |..._Z5minusii...|
00000080  80 80 80 00 04 84 80 80 80 00 00 41 00 0b 87 80  |.....A....|
00000090  80 80 00 00 20 01 20 00 6a 0b 87 80 80 80 00 00  |.... .j.....|
000000a0  20 00 20 01 a0 0b 87 80 80 80 00 00 20 00 20 01  |..... .|
000000b0  6b 0b  |k.|
000000b2

```

图 2-50 hexdump 命令的执行输出信息

下面我们根据之前介绍过的宏观层面的模块组成结构，来分析模块在二进制数据“微观”层面的具体组成形式。

我们需要回忆一下之前介绍的关于 Wasm 模块二进制编码的基础知识。Wasm 模块中的二进制数据是以小端模式进行存储的，因此对于每一个有独立功能的二进制代码段，其二进制

数据的 LSB 位（最低有效位）都存在于该段内容的首字节中。接下来，我们将会按照图 2-50 所示的模块数据从内存低位（0x0）到内存高位（0xb0）的排列顺序来进行分析。

魔术代码（Magic Number）

[00 61 73 6d]

首先，每一个符合标准的 Wasm 模块都是以一段 unit32 类型的“魔术代码”作为模块的起始字节的。魔术代码本身并没有实际的逻辑功能，但是它却是构成一个标准 Wasm 模块所必不可少的部分。由于魔术代码的数据类型为 unit32，因此它占用模块起始部分的 4 个字节，根据小端模式将这部分数据的顺序进行相应的调整便可以得到魔术代码的实际值。如果将该值以十六进制形式表示，则为 0x6d736100，这便是每个 Wasm 模块都必须拥有的结构。因此，我们说一个体积最小的标准 Wasm 模块其内容便是只有这段魔术代码——它可以没有任何其他的模块结构，但是魔术代码却是必不可少的。

如果将魔术代码以每一个字节为一个字符的方式编码成对应的 ASCII 字符串，你会发现这段二进制序列正对应着一段内容为“\0asm”的字符串。

版本号（Version）

[01 00 00 00]

紧接着魔术字段的的是类型同样为 uint32 的模块版本号，该字段指出当前 Wasm 模块在创建时基于的 WebAssembly 标准版本。在当前的 MVP 标准中，该字段被设置为“1”，即对应于十六进制数值“0x1”。

接下来的部分是由众多的“段”结构组成的，这些段结构在组成方式上有着统一的整体格式和字段排列顺序。每种类型的段结构都会有一个全局唯一的名为“id”的字段，用来标识其自身的段类型和对应功能。包括自定义段和 Wasm 标准类型段在内的每一种类型的段结构，其对应的实际二进制段结构都会按照表 2-1 所示的字段组成方式来组织其自身内容。

表 2-1 段结构的二进制数据组成方式

字段名称	类 型	描述信息
id	varuint7	用于标识段类型
payload_len	varuint32	段内容长度
name_len	varuint32 ?	段名长度（用于自定义段，即 id 为“0”）
name	bytes ?	段名内容（用于自定义段，即 id 为“0”）
payload_data	bytes	段具体内容

接下来，我们将按照表 2-1 所示的段结构二进制数据组成方式来依次对模块中剩余的二进制字节数据进行分析。首先，紧接着模块版本号的一个字节数据便是模块中第一个段结构的 id 字段。这里我们将该字节数据 “[01]” 按照 varuint7 的数据编码方式进行相应的解码，并将其转换成对应的十进制数据，然后再通过下面给出的段结构 id 字段值与类型对照表 2-2，便可得到该段结构的实际段类型。

表 2-2 段结构的类型标识符

段结构类型	id 字段值	描述信息
Type	1	模块内函数的签名定义
Import	2	模块导入信息的定义
Function	3	模块内函数定义
Table	4	模块表结构定义
Memory	5	模块线性内存段定义
Global	6	模块全局变量定义
Export	7	模块导出信息的定义
Start	8	模块初始化钩子函数定义
Element	9	模块段结构定义
Code	10	模块内函数的函数体定义
Data	11	模块数据段定义

可以看到，在当前的 MVP 标准中一共定义了 11 种标准的 Wasm 二进制段结构类型。在模块的二进制格式组成方面，我们只需要通过将上述这些标准微观层面上的段结构类型进行相应的组合使用，便可以构建出模块宏观层面上的各种段结构。比如对于接下来要介绍的模块二进制数据组成中的 Type 段、Code 段以及 Function 段，通过将这三个二进制类型段在其各自功能的基础上进行相应的组合使用，便可直接模拟出对应模块宏观层面上 function/code 段所描述的功能和作用。

紧接着 id 字段的是类型为 varuint32 的 payload_len 字段，该字段主要用来标识当前段结构的内容长度。通过计算和占位符的个数可知，该字段占用了模块二进制数据中接下来的 5 个字节 “[... 91 80 80 80 00 ...]”。我们将这 5 个字节按照 LEB-128 的编码方式进行相应的解码操作，便可得到该字段的十进制值 “17”，即该段结构的所有二进制数据将占用 17 字节大小。

这时可能有人会问，段结构中 payload_len 字段的内容在 varuint32 编码下明明可以使用更少的字节数来进行存储，但为何这里却占用了 5 个字节的存储空间？这是由于 Emscripten 等编

译工具链在实际编译 Wasm 模块的开始阶段，并不知道模块中各个段结构所占用的实际字节大小。另外，编译过程必须要按照字节递增的顺序进行，因此编译器只能在这些模块的前置字段中预先留下充足大小的占位字节，待后续得知段结构的真实长度后再进行“数据修正”。而这里提到的“充足大小”便是指对应字段数据类型的最大可用字节数。比如对于一个 varuint32 类型的字段，其最大可用字节数便为向上取整的“32 / 7”（编码时以 7 位为一组），即 5 个字节。

接下来，在 payload_len 字段后面紧跟着的 name_len 和 name 两个字段主要用于标识非标准段结构的别名。非标准段结构即自定义段结构，一个自定义段结构的 id 字段值需要被设置为“0”，因此只能通过这两个与别名相关的字段来标识自定义段的功能和描述信息。关于自定义段这里不做过多的介绍，其详细内容可以参考官方文档。

在组成二进制段结构数据的所有字段中，最后一个字段为 payload_data，在该字段中存放着每一种段结构的核心数据，并且该字段实际存储的内容也随着段结构类型的不同而不同。比如在当前 Wasm 模块中，我们遇到的第一个段类型为 Type 段(id 值为 1)，因此存储在 payload_data 字段中的内容会按照 Type 段规定的字段格式来进行存储。

Type 段（Type Section）

在一个 Wasm 模块的二进制代码中，Type 段的主要作用是用来保存所有导入该模块或直接定义在模块内的函数的函数签名类型（由函数的形式参数和返回值类型共同决定）。Type 段与其他类型的段结构一样，有着自己特定的数据字段组成，以及存储格式要求，如表 2-3 所示。

表 2-3 Type 段类型的字段组成结构

字段名称	类 型	描述信息
count	varuint32	函数签名实体的个数
entries	func_type*	函数签名实体（多个）

Type 段结构本身也是由几个具有不同功能的字段组成的。其中第一个字段为“count”，该字段主要用来标识 Type 段结构中存储的函数签名个数，其二进制数据编码类型为 varuint32。第二个字段为“entries”，在该字段中，模块按照特定的字段排列格式存储着多个 func_type 类型的函数签名实体。而 func_type 类型本身又是由表 2-4 所示的这些特定的字段组成的。虽然可以在 entries 字段中包含任意多个函数签名实体的数据内容，但该字段中函数签名实体的总个数需要与 count 字段的值保持一致。

表 2-4 func_type 实体类型的字段组成结构

字段名称	类 型	描述信息
form	varint7	类型构造符 “func” 对应的 OpCode 值
param_count	varuint32	函数定义的形式参数个数
param_types	value_type*	函数形式参数类型（由类型构造符表示）
return_count	varuint1	函数的返回值个数
return_type	varlue_type?	函数的返回值类型（由类型构造符表示）

我们再把目光移回到模块的二进制数据上。这里首先需要知道该 Type 段结构中函数签名实体的个数，而该字段对应的二进制数据便是接下来模块数据中的一个十六进制字节 “[`… 03 …`]”。我们将该字节按照 `varuint32` 的编码方式进行解码，便可得到函数的签名实体个数“3”，即模块在 Type 段结构中定义了三个函数的函数签名类型。

接下来，我们便可以通过将类型构造符“func”对应的 OpCode 值（`0x60`）作为分割特征值，来寻找在 `entries` 字段中定义的三个函数签名实体。这里将以其中的一个签名实体为例来进行介绍。通过特征分割我们可以发现，在接下来的模块二进制数据中，[`… 60 00 01 7f …`] 4 个字节组成了第一种类型的函数签名实体，随后的 [`… 60 02 7f 7f 01 7f …`] 6 个字节组成了第二种类型的函数签名实体，最后的 [`… 60 02 7c 7c 01 7c …`] 6 个字节组成了第三种类型的函数签名实体。函数签名实体的个数不直接与模块内可用函数的个数相对应，两个不同功能的函数可能拥有完全一致的函数签名实体。下面我们来看最后一个函数签名实体中的数据内容。

首先，该签名实体数据中的第一个字节值为“`0x60`”，即类型构造符“func”对应的十六进制 OpCode 值，这也是构成 `func_type` 数据类型的第一个必需字段。第二个字节值“`0x02`”指出了当前函数签名拥有的形参个数，这里为“2”。紧接着“形参个数”字段的是通过类型构造符描述的形参类型说明符，这里函数签名中的两个形参均对应着 OpCode 值为“`0x7c`”的 `f64` 类型，即两个 64 位浮点数。函数的形参签名部分被定义完成后，接下来便是函数的返回值签名部分。与形参签名部分一样，接下来的一个字节值“`0x01`”指出了函数的返回值个数（在 MVP 标准中子函数/子程序最多只能有一个返回值），因此这里默认为一个返回值。最后的字节值“`0x7c`”同样以类型构造符的方式指出了函数的返回值类型，即一个 64 位的浮点数。这便是 `func_type` 数据类型的完整组成方式。

Table 段（Table Section）

其实对于模块二进制数据中剩余部分的所有段结构类型，我们都可以按照上述方式来进行数据分析，因此这里不再按照顺序分析剩下的模块内容。取而代之的是，我们会挑选一些较为

特殊的段结构来进行介绍。比如位于模块内容“0x2a”到“0x33”的 10 个字节数据 [04 84 80 80 80 00 01 70 00 01] 组成了模块二进制数据中的 Table 段结构。与 Type 段结构一样，在这段二进制数据中，第一个字节的数据值标识了该段结构的具体类型，即十六进制数据“0x4”为模块中 Table 段结构的标识编码。Table 段类型的字段组成结构如表 2-5 所示。

表 2-5 Table 段类型的字段组成结构

字段名称	类 型	描述信息
count	varuint32	Table 段结构中实体的个数
entries	table_type*	Table 实体数据（可能存在多个）

总的来说，在 Wasm 模块的二进制数据组成结构中，所有段结构都是由类似的多个对应段类型的数据实体（如 table_type 类型、func_type 类型等）组成的。在段结构的头部信息中，记录着该段结构中类型实体的个数，而段结构本身是否可以存储多个类型实体则由 WebAssembly 标准来决定。如果将整个段结构比作特定类型数据的存储容器，则段结构中众多的类型实体便组成了模块内的资源索引空间结构（从宏观层面来看）。现在我们回过头来看这个 Table 段结构的组成方式。Table 段结构的段类型实体由 table_type 实体类型组成，table_type 实体类型的字段组成结构如表 2-6 所示。

表 2-6 table_type 实体类型的字段组成结构

字段名称	类 型	描述信息
element_type	elem_type	在 Table 实体中可以存放的数据类型
limits	resizable_limits	Table 实体的描述信息

从表 2-6 中可以看到，每一个 table_type 实体类型又由两个不同的字段组成。第一个字段 element_type 用来标识在该实体类型中可以存放的 Table 数据类型。在 MVP 标准中，这里仅可被标识为“anyfunc”这一种类型，即带有任意函数签名的 Wasm 函数引用。第二个字段“limits”用来记录该实体类型的长度描述信息，而该字段对应的 resizable_limits 类型本身又是由另外一些特定的字段数据组成的，如表 2-7 所示。

表 2-7 resizable_limits 类型的字段组成结构

字段名称	类 型	描述信息
flags	varuint1	1，设置了最大表空间参数；0，没有设置
initial	varuint32	初始的表空间个数
maximum	varuint32 ?	最大表空间个数（若 flags 字段值为“1”）

上述字段主要用来描述模块中表结构实体的长度，即在某个表结构实体中可用表空间个数的相关信息。第一个字段“**flags**”标识了该表结构实体在初始化时是否被指定了可用的最大表空间个数，该字段是一个以 **varuint1** 类型表示的布尔值，若值为“1”，则表示条件成立。第二个字段“**initial**”表示该表结构实体在初始化时指定的可用表空间个数，该字段为 **varuint32** 类型。第三个字段“**maximum**”用来标识该表结构实体的最大可用表空间个数。该字段是否存在取决于 **flags** 字段的实际值（通常来讲，比如我们在 Web 宿主环境中通过 **WebAssembly.Table** 构造函数创建表结构时，其表结构描述符中的 **maximum** 属性为可选项）。若 **flags** 字段的值为“1”，则 **maximum** 字段便存在，该字段的数据存储类型为 **varuint32**。

接下来，我们将参照上面介绍的 **Table** 段数据组成结构来分析实际的模块二进制数据。在此之前，我们已经得到了模块 **Table** 段结构的完整二进制数据部分，这部分数据一共包含 10 个字节。第一个字节为标识具体段类型的标识符；从第二个字节到第六个字节共 5 个字节的数据 [... 84 80 80 80 00 ...] 组成了用来描述段数据长度的 **payload_len** 字段，在这部分数据中含有用来占位的“0x80”字符。我们将这部分数据进行解码便可得到对应的段数据实体长度，即 **Table** 段结构的核心数据部分（**count** 字段与 **entries** 字段）共占用 4 个字节的存储长度；第七个字节数据 [... 01 ...] 表示 **Table** 段结构中定义的 **table_type** 实体个数，这里的结果为“1”。从第八个字节到第十个字节共 3 个字节的数据 [... 70 00 01 ...] 组成了模块 **Table** 段结构中唯一定义的 **table_type** 类型实体。其中首位的“0x70”字节表示可以存放到表结构中的数据类型，该字节以类型的 **OpCode** 编码表示，这里为 **anyfunc** 类型。第二个字节“0x0”对应着 **flags** 字段的值，该值标识了表实体在进行初始化时并没有指定其可用的最大表空间个数。最后一字节数据“0x1”表示表结构初始化时的可用表空间个数，这里为“1”。

Elem 段（Element Section）

我们之前介绍过，模块利用 **Table** 段来生成用于存放不透明数据的容器，这些被存放在 **Table** 段中的不透明数据可以被模块上层宿主环境中的特定接口使用。当 **Table** 段创建完毕后，还需要在模块中通过名为“**Elem**”的段结构来为 **Table** 段中的表空间填充数据。与其他段结构类似，**Elem** 段也是由特定的两部分字段数据组成的，如表 2-8 所示。

表 2-8 Elem 段类型的字段组成结构

字段名称	类 型	描述信息
count	varuint32	在 Elem 段中对应实体的个数
entries	elem_segment *	Elem 实体数据（可能存在多个）

与模块中的 **Table** 段一样，**Elem** 段也同样是由多个段实体结构组成的，每一个段类型实体

都直接对应一次 Table 段中表空间的数据填充操作。其中第一个字段“count”标识了在该 Elem 段结构中段实体的个数；第二个字段“entries”作为段实体数据的存入口位置，是由众多的段类型实体数据按照顺序依次重复排列而成的。段实体类型“elem_segment”的字段组成结构则如表 2-9 所示。

表 2-9 elem_segment 实体类型的字段组成结构

字段名称	类 型	描述信息
index	varuint32	要填充的 Table 实体索引（在 MVP 标准中为 0）
offset	init_expr	计算偏移位置的常量表达式
num_elem	varuint32	要填充的元素个数
elems	varuint32 *	要填充的元素数据（在 MVP 标准中为函数索引）

每一个 Elem 段实体类型 elem_segment 的数据部分都是由 4 个字段组成的。第一个字段“index”指定该实体将要进行数据填充的 Table 类型实体在 Table 段中的索引位置，在 MVP 标准中，该字段仅可以取值“0”，即对应模块中默认仅有的一个 Table 类型实体。第二个字段“offset”是由一个标准的 WebAssembly 常量表达式组成的，该字段需要基于一个 i32 类型的常量表达式来计算将要进行数据填充的表空间偏移位置。第三个字段“num_elem”为将要进行填充的数据个数，每一条数据都对应一个独立的表空间。最后一个字段“elems”表示将要填充的数据实体。由于在当前 MVP 标准中规定，只允许在 Table 段结构中填充类型为“anyfunc”的数据，因此这里的 elems 字段仅允许存放对标准 Wasm 函数的引用类型。

接下来，我们将在模块的二进制数据中分析该模块 Elem 段的实际组成结构。在模块内容中，从位置 0x77 到 0x83 共 13 个字节的数据 [⋯ 09 87 80 80 80 00 01 00 41 00 0b 01 01 ⋯] 组成了该模块的 Elem 段结构。首字节的数据“0x9”为模块 Elem 段的标识编码；从第二个字节到第六个字节共 5 个字节的数据 [⋯ 87 80 80 80 00 ⋯] 标识了该 Elem 段结构的数据总长度，这里经过解码后可以得知该段结构的数据长度为 7 个字节；第七个字节 [⋯ 01 ⋯] 标识了在模块 Elem 段结构中定义的段实体个数，这里为“1”；第八个字节 [⋯ 00 ⋯] 为 Elem 段结构将要进行数据填充的 Table 类型实体索引值，这里默认为“0”；从第九个字节到第十一个字节共 3 个字节的数据 [⋯ 41 00 0b ⋯] 组成了一个常量表达式。该常量表达式指定了 Elem 段结构将要进行数据填充的表空间偏移位置。这里返回了一个值为“0”的位置，即对应于表结构实体的初始位置，不进行任何偏移。第十二个字节的数据 [⋯ 01 ⋯] 指定了 Elem 段结构将要存放的不透明数据个数，这里为“1”；最后一个字节的数据 [⋯ 01 ⋯] 表示将要存储的实际数据内容，在 MVP 标准中规定只能存放标准 Wasm 函数的引用。这里指定了将要存储的数据为模块中对应函数索引空间中位置为“1”处的函数引用（即 WAT 中标记名为“\$Z3addii”的函数）。

至此，我们已经从模块整体的宏观层面，以及深入到模块二进制数据内部细节的微观层面了解了一个 Wasm 模块的具体组成结构。但无论是从宏观还是微观角度来看，众多不同类型的段结构是组成一个标准 Wasm 模块的最基本元素。每一种类型的段结构都以其各自的方式记录着标准模块所需要的不同类型的数据结构，这些段结构在模块的各个生命周期内发挥着不同的作用。Wasm 模块在宏观和微观层面的组成方式分别如图 2-51 和图 2-52 所示。

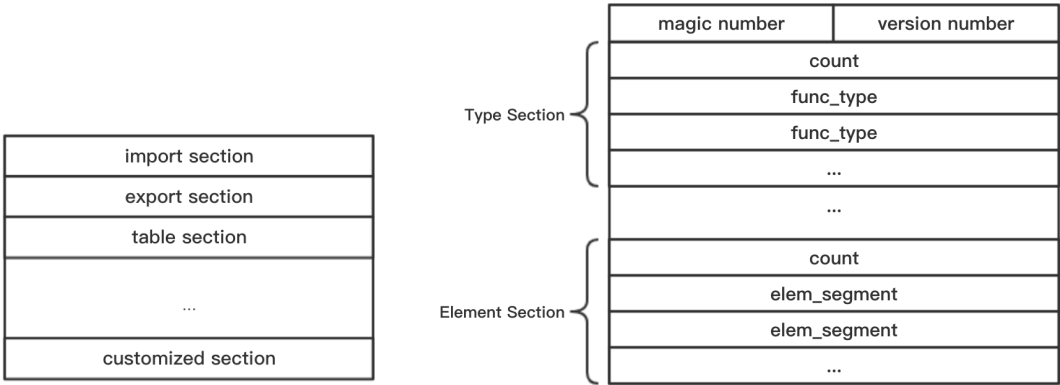


图2-51 Wasm模块在宏观层面的组成方式

图2-52 Wasm模块在微观层面的组成方式

从宏观层面来看，Wasm 模块主要由 10 种标准段结构和一些自定义段结构组成，整体组成结构十分简单明了。

从微观层面来看，Wasm 模块由魔术字符（数字）、版本号和在标准中定义的 11 种二进制段结构组成。虽然模块在微观层面定义的段结构类型与在宏观层面的段结构类型并不是一一对应的，但是这 11 种二进制段结构却从模块的微观层面构建出了宏观层面上各种类型段结构对应的特定功能。

2.6 内存结构

在前面的章节中，我们曾介绍过 Wasm 抽象虚拟机在实例化模块时对模块内存资源的分配模式及数据处理方式。本节我们将继续深入了解这方面的内容。

模块中的共享线性内存段是一个地址连续的、可以进行字节寻址的内存结构，模块可以从该结构的正整数索引地址“0”处开始向高地址进行数据寻址操作。由于存在分页机制，内存段的可用大小将一直保持为一个 WebAssembly 内存页对应内存大小的倍数。在 MVP 标准中，一个 WebAssembly 内存页被规定为固定的 64KB 大小。模块在初始化时其共享线性内存段的状态

由模块内定义的 **Memory**（定义共享内存段的初始大小、最大可用大小等属性）段和 **Data** 段结构（定义共享内存段中的初始数据）来共同决定。

在共享线性内存段中模块仅存储最原始的二进制比特数据，而数据对应的实际值类型则由模块中具体的虚拟指令逻辑来决定。整个线性内存段被放置在一个完全封闭的沙箱环境中，这使得模块的共享线性内存段与浏览器中其他类型的线性内存完全分离。不能被模块直接触及的内存类型包括：抽象虚拟机引擎的内部数据结构、数据栈容器所使用的内存、在 **PM** 空间中声明的局部变量、代码段数据，以及其他应用程序进程使用的内存等。总的来说，虚拟机提供的沙箱环境可以让模块内部的线性内存段被更加安全地使用。

2.6.1 操作运算符

在 **WebAssembly** 标准中规定了一系列可以用来对共享线性内存段数据进行读取与写入操作的虚拟指令，这些指令可以将特定内存位置处的原始比特数据转换成特定类型的数据值，同时也可以将这些数据值以小端模式再写入指定位置的内存单元中。如下列举出一些专门用于对内存段进行数据操作的虚拟指令，其他没有列出的内存操作虚拟指令与这些指令的功能类似。

i32.load

从内存的指定索引位置处读取 4 个字节的数据，并将其转换为一个 **i32** 类型的数值，即一个 32 位整数。

i32.load16_s

从内存的指定索引位置处读取 2 个字节的数据，并将其通过“0 值占位填充”的方式转换为一个 **i32** 类型的数值，即一个 32 位整数。

i32.store

向内存的指定索引位置处以 **i32** 类型存储一个数值，该数值会占用 4 字节的存储空间。

i32.store16

向内存的指定索引位置处以 **i16** 类型存储一个 **i32** 类型的数值，该数值会占用 2 字节的存储空间。当该 **i32** 值的二进制数据长度超过 **i16** 类型的最大数据长度时，该数值会被截断。

2.6.2 寻址

对于每一个用于操作（读写）共享线性内存段的虚拟指令，模块在调用它们时都需要为其指定内存段的入口地址，也就是共享内存段中某个具体字节位置对应的索引值，而虚拟指令则

会从内存的该位置处开始进行数据的读写操作。我们之前介绍过，基于内存的分页机制，一个应用程序的内存逻辑地址由两部分组成：第一部分为一个指定了具体内存页的页号，该参数指定了目标内存地址所在的内存页；第二部分为一个用于确定目标地址在该内存页中具体偏移位置的偏移量。在 Wasm 模块中，我们将某个数据被存储在共享线性内存段的实际索引位置称之为该数据在线性内存段中的“有效地址（Effective Address）”，其组成方式与逻辑地址类似，但略有不同。模块在进行方法调用时会直接将内存的有效地址解释为从共享内存段的首字节到目标字节的相对偏移量，该值是一个无符号整数。当我们以可读文本代码（WAT）来表示模块中内存操作部分的指令逻辑时，被放置在 load 等指令后用于表示具体内存索引位置的参数值，便是对应的共享内存段首字节位置的相对偏移量索引值。

实际上，在当前的 MVP 标准中，模块可以对共享内存段进行的最大可寻址长度为 32 位（对应内存偏移量索引值的最大值），即对应着 Wasm32 体系的 4GB 最大可用共享线性内存。后续将要发布的 Wasm64 系列标准将允许对最长为 64 位的共享线性内存段地址进行数据寻址操作，这样模块的可用线性内存容量将会大大超过现有的 4GB 大小。

2.6.3 对齐

我们知道，传统的计算机是由多种类型的总线结构组成的。这些总线结构负责将计算机中的运算单元、存储单元和输入输出单元等各类部件进行连接，进而使得它们可以通过总线结构来传递数据和指令等各类信息。在所有的总线结构中，“数据总线”负责向 CPU 传输在运算过程中需要使用的数据，而“地址总线”则负责对应的内存地址中读取数据。

在一个基于 x86 处理器架构的计算机中，由于其 CPU 的字长为 32 位，因此该计算机的数据总线与地址总线宽度也均为 32 位。这就意味着当地址总线从内存中读取数据时，在每一个 CPU 的时钟周期内都会从内存中的某个地址直接读取对应长度为 32 位的数据，经过换算即 4 字节大小的数据。同理，数据总线也会在每一个 CPU 的时钟周期内向 CPU 发送 4 个字节大小的数据用于运算，这也是 CPU 在每一次时钟的震荡周期内所能够处理的最大数据总量。虽然在计算机内部这种传统的数据“交流方式”看似没有任何问题，但是由于其地址总线与数据总线每次所能够读取的数据长度是固定的，便可能导致一系列 CPU 在内存数据读取上产生的不必要的性能损耗，因此有必要对内存数据的具体存储方式进行一些调整。

一般来说，在 C/C++ 等强类型语言中，我们需要在声明变量时显式指定该变量所属的具体类型，而不同类型的变量占用不同大小的存储空间。当 CPU 通过地址总线从某个具体的内存地址读取数据时，如果该数据能够正好在 CPU 的单一时钟周期内被完全读取出来（即数据正好完全处于该次内存读取的有效地址段内），则此时内存数据的读写效率是最高的。但实际上，由于

应用程序中各种数据类型变量占用的内存空间不尽相同，因此未经特殊对齐处理的数据有可能需要占用两个甚至两个以上的 CPU 时钟周期才能够被完全从内存中读取出来，而此时应用程序的整体运行效率便会十分低下。比如在下面给出的 C++ 结构体代码中，假设该结构体中各元素对应的实际内存数据没有经过对齐处理，而是直接以连续内存地址的方式进行存储的。结构体的定义代码如下。

```
struct S {  
    bool a; // 1  
    int b; // 4  
    bool c; // 1  
    double d; // 8  
    bool e; // 1  
};  
  
int main (int argc, char* argv[]) {  
    struct S s = {true, 10, false, 10.1, true};  
    return 0;  
}
```

在上面的代码中，我们声明了一个名为“S”的结构体，在该结构体中定义了 5 个不同类型的元素。我们将每个元素所对应类型占用的存储空间大小，以注释方式标记在了这些元素定义语句的后面。接下来，假设在内存中使用连续地址的方式来存储这些数据，那么此时该结构体在经过初始化后内存段中各元素数据所在的地址及相对位置如图 2-53 所示。

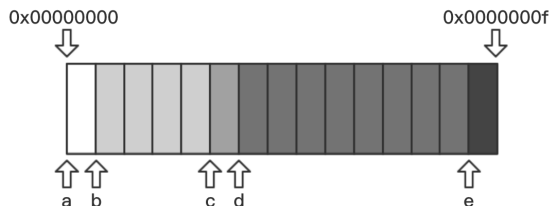


图2-53 结构体S中各属性值在内存中的相对存储位置（连续存放）

可以看到，以连续地址来存储数据，可以最大限度地提高内存空间的使用效率。在这块内存空间中，结构体对象内的元素按照顺序依次占用对应数据类型大小的存储空间。在大多数现代计算机系统中，地址总线每次只能从固定位置的内存地址处进行数据读取操作。比如在一个基于 x86 架构的计算机系统中，由于其地址总线的长度为 32 位，因此 CPU 只能通过地址总线从内存地址为 4 的倍数（0x0 除外）的偏移地址处进行数据读写操作。而对于上述结构体，当在线性内存中按照连续地址的方式来存储数据时，结构体中元素“b”的数据值便需要经过两个

CPU 时钟周期才能被完全读取出来。如图 2-54 所示，在第一个时钟周期内，地址总线会读取内存偏移地址从 0x0 到 0x3 共 4 个字节的数据，然后 CPU 会通过相应的位移运算来获取数据中属于元素“b”的部分并将其暂时缓存起来。在第二个时钟周期内，地址总线会读取内存中从偏移地址 0x4 到 0x7 共 4 个字节的数据，并随后将数据中属于元素“b”的部分再次通过位移运算取出。最后 CPU 会将从这两个时钟周期内取出的属于元素“b”的两部分数据进行组合，然后才能够得到元素“b”的最终数据值。

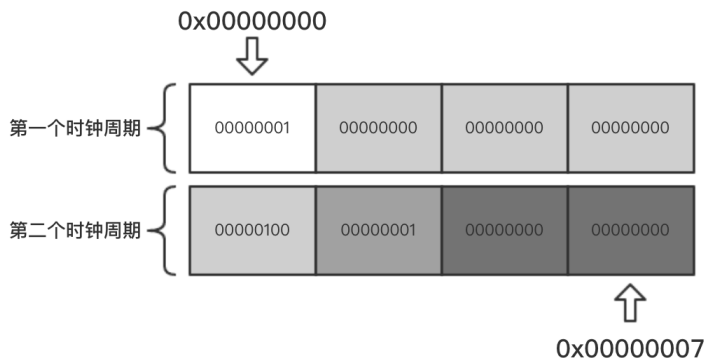


图2-54 数据被分散在两个CPU时钟周期内的寻址情况

不仅如此，在整个数据寻址和位移运算过程中，还会涉及各类 CPU 指令集的调用，以及相关总线的读写操作，而这无疑会大大降低 CPU 的数据处理效率。为了能够适应各类底层 CPU 架构的地址总线读写规则，同时使内存数据的读写效率达到最高，我们需要对存放内存中的数据进行处理，以便让那些被经常使用到的数据能够尽可能的在一个 CPU 时钟周期内就从内存中读取出来并交给 CPU 进行处理。

常用的数据对齐方式有很多种，这里只介绍其中最常用的一种，即“自然对齐”。所谓自然对齐，即数据在内存中进行存储时的起始内存地址必须为对应数据类型占用内存大小的整数倍（均以字节为单位进行计算）。比如在 C/C++ 语言中，通常一个 double 类型的数据占用的存储空间为 8 个字节，因此当在内存中以自然对齐的方式来存储该数据时，便需要将其存储在起始内存偏移地址为“8”的整数倍的地址上，比如 0x8、0x10 等可以被十进制数 8 完全整除的内存地址。既然需要对内存中的数据进行自然对齐处理，那么为了能够让特定类型的数据被存放在符合特定规则的内存地址上，我们会在实际存放数据时在这些不同类型数据对应的内存数据中填充用于占位的空数据来实现数据对齐的效果。比如对于上面的 S 结构体，当采用自然对齐的方式来存储该数据结构时，结构体内的各元素在实际内存中的存储状态如图 2-55 所示。

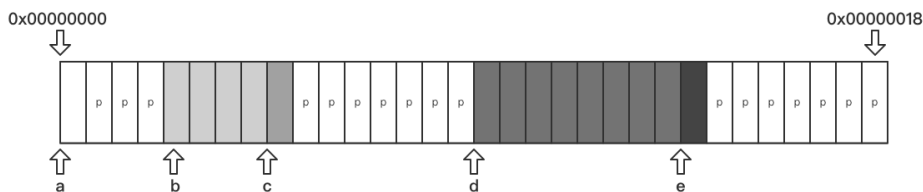


图2-55 结构体S内各元素在实际内存中的存储状态（经过对齐）

可以看到，经过自然对齐处理后的结构体数据其元素值在实际内存中的存储位置发生了改变。比如通过在元素“a”与元素“b”之间填充 3 个占位数据的方式，让元素“b”的存储地址能够满足自然对齐的要求，即存放数据的首地址为对应数据类型所占用存储空间大小的整数倍（这里为 4 个字节的数据对应的索引值为 0x4 的内存地址）。同理，位于元素“c”与元素“d”之间的 7 个占位数据，也使得元素“d”能够存放在起始地址为“0x10”的内存地址处，这让元素“d”的数据存储方式满足了自然对齐的要求。这里需要注意的是，位于元素“e”后面的 7 个占位数据主要用来让整个结构体数据能够保持统一的大小，从而使 CPU 可以通过一致的位移及数据处理操作，来获取一个结构体数组中各结构体对象的元素数据。从某种程度来讲，我们也可以将其理解为对结构体数据整体进行的数据对齐处理。

通常来说，在类似 C/C++等语言中，编译器会自动对在程序中定义的结构体、联合体甚至是一些常用的基本数据类型进行数据对齐处理。在一个 Wasm 模块中，我们也可以通过修改可读文本代码（WAT）的方式，来主动干预模块在读写共享线性内存段数据时进行的数据对齐处理。我们可以通过在 store 等与内存操作相关的虚拟指令中加入 align 和 offset 字段来指定对数据进行的对齐处理。其中 align 字段主要用来标记该操作数是否进行了对齐处理，并且显式指出按照怎样的方式（对齐字节数）进行数据对齐；offset 字段则主要用来对数据的具体存储地址进行适当的位置偏移，以满足 align 字段指定的数据对齐要求。在介绍这两个用于进行内存数据对齐的特殊字段前，我们先来了解一下“有效地址”的概念。

在 MVP 标准中规定，一个数据的有效地址是指在通过某个线性内存数据运算符对其进行读写操作时，该数据在内存段中的实际存放地址。换句话说，有效地址也可以用来指代某个线性内存数据运算符在经过数据对齐处理后所指向的真实目标地址。接下来，我们将通过如下 Wasm 可读文本代码来进一步解释有效地址的概念。

```
...
(i32.store offset=2 align=4 (i32.const 10) (i32.const 1))
...
```

在上面这段代码中，我们通过 i32.store 这条虚拟指令在线性内存段的索引位置“0”处写入

了一个值为“1”的 32 位整数。同时还通过 `align` 字段标记出在线性内存中存放该数据时是以 4 个字节为宽度来进行数据对齐处理的。`offset` 字段指出了数据的实际存储位置与该虚拟指令操作数给定位置的偏移距离，这里给出的偏差距离为“2”。至此，我们便可以得到数据在内存中的实际存放地址，即数据的有效地址为内存的索引位置“12”（操作数地址“10”+ 偏移地址“2”），同时该索引地址也正好满足以 4 个字节为大小的数据对齐条件。此时该数据在共享线性内存段中的对齐存储方式如图 2-56 所示。

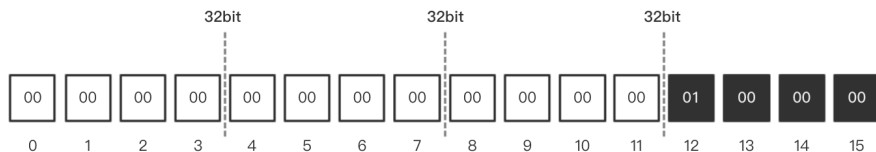


图2-56 数据在Wasm模块共享线性内存段中的对齐存储方式

可以看到，该 32 位整数以小端模式的 LEB-128 编码形式被存放在共享线性内存段的第 12~15 个索引位置上。这使得该整数的内存数据存储方式正好满足自然对齐的条件，即数据在内存中存储位置的首地址正好为该数据类型所占存储空间的整数倍。实际上，是否使用这种通过 `align` 字段在模块中显式标记数据对齐的声明方式，并不会影响整个 WebAssembly 标准的语法和语义结构。只不过对于具体的 Wasm 底层虚拟机实现来说，这种显式标记数据对齐的声明方式可以让虚拟机在数据读取层面对模块代码进行优化。当然，如果我们在模块代码层面显式地声明了 `align` 字段标记的数据对齐方式，但数据在内存中的有效地址并没有满足数据对齐的条件（比如没有通过 `offset` 字段对首地址进行偏移处理），则此时底层虚拟机在执行模块中数据的读写操作时其效率会大大降低。

在当前的 MVP 标准中，规定 `align` 字段的对齐字节数必须为 2 的整数次幂，并且该值不能大于线性内存数据运算符所代表类型的数据宽度。比如 `load.i32` 指令对应的 `i32` 类型其数据宽度为 4 个字节，那么该内存数据运算符可以被显式指定的数据对齐字节数不能超过 4 个字节，即 `align` 字段值不能大于 4，其他数据类型同理。而当指定的对齐字节数与该数据类型的数据宽度相同时，我们便认为此时为数据的自然对齐状态，即在线性内存中对该数据进行读写操作时效率达到最高。

事实上，Wasm 模块的底层虚拟机根据其上层宿主环境的不同而有着不同的实现方式。为了让数据在各类底层虚拟机的实现上都能够最大限度地保持最高的读写效率，我们建议对那些需要经常进行读写处理的内存数据进行模块层面的自然对齐处理，并通过 `align` 字段进行显式标记。这样一旦模块宿主环境的底层虚拟机支持对模块内部标记了数据对齐的代码进行优化，我们就可以将模块中围绕该数据的处理逻辑指令的执行效率提升到最高。

2.6.4 溢出与调整

在当前的 Wasm32 体系中，一个共享线性内存段的最大可用长度（容量）为 4GB，当我们在模块中通过与线性内存相关的虚拟指令来操作一个越界的内存地址时，模块便会直接产生一个异常，并中断当前指令段的执行流程。接下来模块会将该异常向上层的宿主环境中抛出。一般来说，当使用浏览器作为 Wasm 模块的外部使用宿主环境时，从模块中抛出的异常会在宿主环境中直接转换为对应的 JavaScript 异常对象。

在当前的 MVP 标准中，还提供了两个能够操作共享线性内存段的特殊虚拟指令，如下所示。通过使用它们，我们可以在模块中随时动态地查看当前模块的可用线性内存大小，甚至还能够以“WebAssembly 页”为单位来扩增当前线性内存段的可用容量，直至模块 memory 段中定义的最大容量。

grow_memory

该指令主要用于以“Wasm 页”为单位来扩增共享线性内存段的可用容量。可扩增到的最大容量不能超过模块线性内存段在定义时指定的最大可用容量。

current_memory

该指令主要用于获取当前模块的可用线性内存段容量，该容量会以“页”为单位进行表示。

在当前的 MVP 标准中，共享线性内存段所暴露出的外部可用特性及相关虚拟指令并不多，因此我们只能将线性内存段当作一个简单的用于数据存储和交换的容器。而包括“模块多内存段”、非连续的“保护性内存段”，以及用于释放模块内存段空间的虚拟指令在内的一系列高级特性，都会在将来的 Post-MVP 标准中逐渐出现。

第 3 章

动态链接与 SIMD（基于 MVP 标准）

在第 2 章的内容中，我们主要围绕 WebAssembly 标准中提出的一些基本特性进行了较为深入的理论性介绍。在本章中，我们将继续从理论入手，来了解与 Wasm 相关的更多高级特性与概念，并尝试通过构建几个简单的 Wasm 应用实例来加深对这些抽象概念的进一步理解。总的来说，虽然这些特性并不是 Wasm 独有的，但是其对于了解现代操作系统原理、CPU 高级特性及构建高性能的上层 Web 应用却有着举足轻重的作用。

3.1 动态链接（Dynamic Linking）

我们之前介绍过，计算机在运行一个应用程序时，会将该应用程序所使用到的所有资源全部分配到一段进程独立的 PM（Program Memory）空间中。而在这段 PM 空间中会存放当前应用程序在运行过程中需要使用到的各类标准段，以及其他的附加段结构数据。每一个应用程序所对应的 PM 空间，均是基于该应用程序的独立 VAS（Virtual Address Space，虚拟地址空间）进行创建的。在同一个操作系统中，每一个应用程序的可用 VAS 大小均保持一致。当应用程序运行时，PM 空间中各类段结构数据在实际物理内存中的存储位置，会由操作系统根据当前的系统情况自动进行分配。而数据的实际存储容器可能是当前计算机的物理线性内存，或是硬盘等其他类型的存储介质。

在应用程序运行过程中，其进程所使用到的数据会按照不同的类型被分配到 PM 空间的各个段结构中，其中代码段结构主要负责存储一系列业务逻辑指令和代码。在动态链接技术出现之前，对于大部分应用程序，从整体上看可能会发现这样一个问题，就是在这些应用程序的代码段结构中，某一部分功能的逻辑指令和代码是完全相同的，但是这些指令和代码却重复地出现在每一个应用程序的代码段结构中，并且同时占用应用程序二进制可执行文件大小的一部分。

那么是否有这样一种技术，可以将这些分散在各应用程序中的“通用代码”分离出来，并将它们编译和打包成独立的模块文件呢？这样应用程序便可以在运行时动态加载该模块，并调用其中的这部分通用代码。不仅如此，从另一个方面来看，将通用代码从应用程序中分离，还可以进一步减小应用程序二进制文件占用的存储空间。这种可用于构建和使用包含可重用代码模块的技术，便是我们接下来将要介绍的“动态链接”技术。

由于被分离出的通用代码可以在各个应用程序进程间进行共享，因此我们将这部分通用代码所对应的经过编译处理的二进制模块文件称为“共享库（Shared Library）”。基于每个应用程序所拥有的独立 VAS，我们可以很容易地让多个独立的应用程序进程在运行期间访问位于同一个物理内存页上的“单份”共享库代码，应用程序与共享库之间的宏观交互逻辑如图 3-1 所示。可以看到，两个独立的应用程序进程在其各自的 VAS 中，通过不同的相对虚拟内存地址引用位于同一个物理内存页上的共享数据。

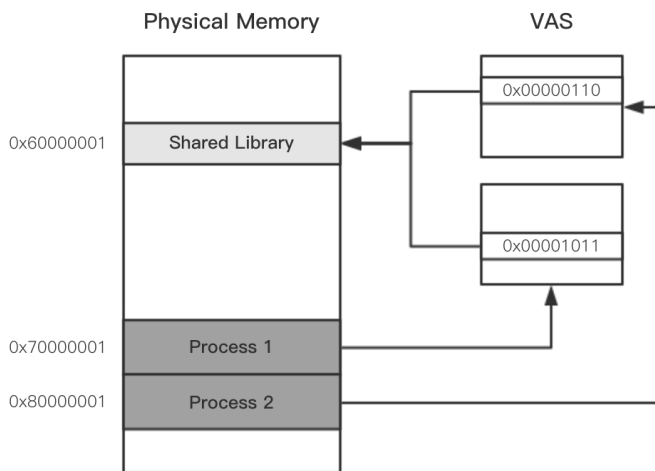


图3-1 应用程序与共享库之间的宏观交互逻辑

实际上，共享库与可执行的二进制应用程序两者内部的段结构是类似的，它们都具有用于存放逻辑指令的代码段，以及用于存放全局或静态变量具体值的数据段。不同的是，共享库并没有提供一个可以用来执行的入口函数地址，相对而言，在它的内部只定义了可以被其他应用程序引用的一系列通用变量及函数等代码逻辑资源。对于大部分基于类 UNIX 实现的操作系统而言，无论是可以直接运行的独立二进制应用程序，还是只能在运行时状态下通过动态链接过程来进行数据引用的共享库，其文件本身均是基于 ELF 这种格式标准进行构建的。下面我们将以 Linux 系统作为实验环境，来深入理解传统动态链接过程的基本实现原理。

3.1.1 ELF

ELF (Executable and Linkable Format, 可执行与可链接格式) 是经常被使用在各类操作系统中, 用于表示二进制文件的一种十分灵活的文件格式标准。ELF 在其标准中规定了一系列在构建二进制格式文件时所必须遵守的字段结构及数据格式等要求。基于统一的标准来构建具有不同功能特性的二进制文件, 使得这些文件可以直接复用现有的 ELF 调试工具, 对其内部的二进制代码逻辑进行调试操作。同时, 基于统一的二进制文件标准, 在某种程度上也使得应用程序在各个操作系统间的代码移植过程变得更加简单。

ELF 头

ELF 从古老的 COFF (Common Object File Format) 格式发展而来, 该格式在发明初期专门用于为 UNIX 系统中的可执行文件、对象文件和共享库文件三种类型的二进制文件制定具体的文件组成格式。相比 COFF 格式而言, ELF 在其基础上又进行了大量的功能及可用性扩展。ELF 允许在其头部信息中自定义任意数量的段结构, 并且允许每一个段结构为其自身附加额外定义的属性信息。这种改进在某种程度上更有利于共享库和二进制应用程序之间进行动态链接。我们可以在类 UNIX 系统中通过如下命令来查看一个标准的 ELF 二进制文件的头部信息。

// 查看系统中 ls 命令对应的可执行二进制文件内所有的 ELF 头部信息

```
readelf --header /bin/ls
```

在该命令的输出结果中, 第一部分内容如图 3-2 所示。

```
[root@iZ23ki8vt8tZ dynamic_linking]# readelf --headers /bin/ls
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                          ELF64
  Data:                             2's complement, little endian
  Version:                          1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:                0x4027e0
  Start of program headers:           64 (bytes into file)
  Start of section headers:          107352 (bytes into file)
  Flags:                              0x0
  Size of this header:                 64 (bytes)
  Size of program headers:            56 (bytes)
  Number of program headers:           8
  Size of section headers:            64 (bytes)
  Number of section headers:          29
  Section header string table index: 28
```

图3-2 通过readelf命令读取二进制文件的ELF头信息

从图 3-2 中可以看到在 ELF 头中定义的所有字段。第一行的魔术字符串字段表示标准 ELF 二进制文件是以该字段对应的二进制字节串作为开头数据的。整个魔术字符串的前四个字节为

固定的数据值，其中第一个字节的值必须为十六进制形式的 0x7f，随后的三个字节数据值分别为 0x45、0x4c 和 0x46。如果将后续这三个字节的数据值按照标准 ASCII 码表中的字符映射关系分别转换成对应的字符实体，则按照转换顺序，可以将它们拼接成一个字母均为大写的字符串“ELF”。如果你还记得标准 Wasm 模块的二进制组成结构，则会发现组成标准 Wasm 模块的二进制数据其开头部分的魔术字符串与 ELF 二进制文件格式的魔术字符串十分相似。只不过当它们对应到 ASCII 码表中的字符时，其中一个会被映射为字符串“ELF”，而另一个会被映射为字符串“\0asm”。

接下来的一系列字段标识了该二进制文件属性的一些相关信息，其中包括：该二进制文件所适用的处理器及操作系统类型、数据的存储形式（二进制补码、小端模式），以及可执行文件的入口函数地址等。这部分用来描述二进制文件综合属性的信息，我们将其称为二进制文件的“ELF 头”。ELF 头从整体的角度描述了该二进制文件内部与 ELF 标准有关的属性信息。整个 ELF 二进制格式的头部信息由两部分组成，第一部分就是包含文件综合属性信息的“ELF 头”；第二部分则是用于描述 ELF 文件具体组成结构的“Program 头”。在该头部给出的内容中，ELF 标准通过 Segment 和 Section 这两种基本数据结构，以结构化的方式描述了二进制文件内各功能段的具体组成信息。

Program 头

在标准的 ELF 格式中，Program 头的主要作用是向操作系统描述二进制可执行文件在运行前需要准备的各类结构化数据。整个 Program 头是由多种不同类型的 Segment 结构描述信息组成的。在标准的 ELF 二进制文件中，每一个 Segment 结构同时又是由另外一种或多种具有不同功能的 Section 结构组成的。因此可以说，具有不同功能的各类 Section 结构是构成整个 ELF 文件数据的最基本结构。如图 3-3 所示的是二进制可执行文件（/bin/ls）头部的部分 Section 结构描述信息，我们也可以称之为 Section 头信息。在这部分信息中描述了 ELF 二进制文件中各类 Section 结构的对应名称、其所在 ELF 文件虚拟地址空间中的偏移地址，以及相关属性信息（具有的权限、类型等）。

在图 3-3 给出的信息中，由第一列序号代表的每两行数据整体上对应于一个 ELF 文件内的 Section 结构。Section 将 ELF 二进制文件内的原始二进制数据流信息整理成具有不同功能逻辑的结构化数据，这些数据将会在程序创建和实例化运行前分别被编译器和链接器两者共享。每一个 Section 结构都具有其可描述性属性，比如该 Section 结构具有的系统权限（操作系统可读、可写及可执行等）、所对应的类型，以及在应用程序虚拟地址空间中该 Section 结构的具体起始地址等信息。如果仔细观察图 3-3，你会发现在这段 Section 头数据的描述信息中含有一个名为

“.text” 的 Section 结构，这个 Section 结构正对应着我们之前介绍过的，存在于各应用程序 PM 空间内的“Text 段”结构。同样的，在这个名为“.text”的 Section 结构中也存放着 ELF 二进制文件的详细代码段。

Section Headers:						
[Nr]	Name Size	Type EntSize	Address Flags Link Info	Offset Align		
[0]	0000000000000000 0000000000000000	NULL	0000000000000000 0 0 0	00000000		
[1]	.interp 000000000000001c	PROGBITS	0000000000400200 A 0 0	00000200		
[2]	.note.ABI-tag 0000000000000020	NOTE	000000000040021c A 0 0	0000021c		
[3]	.note.gnu.build-id 0000000000000024	NOTE	000000000040023c A 0 0	0000023c		
[4]	.gnu.hash 0000000000000064	GNU_HASH	0000000000400260 A 5 0	00000260		
[5]	.dynsym 00000000000000be8	DYNSYM	00000000004002c8 A 6 1	000002c8		
[6]	.dynstr 000000000000005c4	STRTAB	0000000000400eb0 A 0 0	00000eb0		
[7]	.gnu.version 00000000000000fe	VERSYM	0000000000401474 A 5 0	00001474		
[8]	.gnu.version_r 00000000000000a0	VERNEED	0000000000401578 A 6 3	00001578		
[9]	.rela.dyn 000000000000001b0	RELA	0000000000401618 A 5 0	00001618		
[10]	.rela.plt 00000000000000990	RELA	00000000004017c8 A 5 12	000017c8		
[11]	.init 0000000000000018	PROGBITS	0000000000402158 AX 0 0	00002158		
[12]	.plt 00000000000000670	PROGBITS	0000000000402170 AX 0 0	00002170		
[13]	.text 000000000000fb68	PROGBITS	00000000004027e0 AX 0 0	000027e0		

图3-3 二进制可执行文件 (/bin/ls) 的Section头信息

事实上，我们之前在介绍应用程序 PM 空间时给出的五种基本段结构类型，均以 Section 形式描述并被存放在实际的 ELF 二进制文件中。但由于 ELF 本身具有不同于 COFF 的特性，编译器还可以基于当前的五种标准段结构加入其自定义的段结构中，以适用于不同的处理器及编译环境。同样的，这些新增的自定义段结构也会以 Section 的形式进行描述并被存放在具体的 ELF 文件中。比如在大部分 ELF 二进制文件中，我们经常会遇到一个名为“.sdata”的 Section 结构，该结构虽然不属于标准的 PM 空间段结构，但它却会在某些具体类型的处理器架构与操作系统中被编译器用在某些小类型（这里是指单个数据类型占用的存储空间较少，比如字符类型的数据。而对于“小类型”的具体定义则由编译器来进行判断）数据的存储过程上。

如图 3-4 所示便是上述 ELF 二进制可执行文件 (/bin/ls) 的 Program 头信息，以及所有 Segment 与 Section 结构的映射关系。可以看到，在该文件的 Program 头结构中一共定义了 8 个不同类型

的 Segment 结构，在每一个 Segment 结构中又分别含有数量和类型各不相同的 Section 结构。比如这里存放着应用程序源代码的“.text”类型段被分配在第一个类型为“LOAD”的 Segment 结构中，并且该 Segment 结构还同时被指定了“R”与“E”两个标识符对应的系统权限，即表示在应用程序运行时，操作系统可以随时访问在该 Segment 结构中声明的所有 Section 结构其内部的二进制代码数据。

Program Headers:					
Type	Offset	VirtAddr	PhysAddr	Flags	Align
	FileSiz	MemSiz			
PHDR	0x0000000000000040	0x0000000000400040	0x0000000000400040		
	0x00000000000001c0	0x00000000000001c0	R E	8	
INTERP	0x0000000000000200	0x0000000000400200	0x0000000000400200		
	0x000000000000001c	0x000000000000001c	R	1	
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]					
LOAD	0x0000000000000000	0x0000000000400000	0x0000000000400000		
	0x0000000000001851c	0x00000000001851c	R E	200000	
LOAD	0x00000000000019000	0x0000000000619000	0x0000000000619000		
	0x0000000000001240	0x0000000000001f60	RW	200000	
DYNAMIC	0x00000000000019a88	0x0000000000619a88	0x0000000000619a88		
	0x00000000000001d0	0x00000000000001d0	RW	8	
NOTE	0x000000000000021c	0x000000000040021c	0x000000000040021c		
	0x0000000000000044	0x0000000000000044	R	4	
GNU_EH_FRAME	0x00000000000015e88	0x0000000000415e88	0x0000000000415e88		
	0x00000000000006b4	0x00000000000006b4	R	4	
GNU_STACK	0x0000000000000000	0x0000000000000000	0x0000000000000000		
	0x0000000000000000	0x0000000000000000	RW	8	
Section to Segment mapping:					
Segment Sections...					
00					
01	.interp				
02	.interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .				
03	.plt .text .fini .rodata .eh_frame_hdr .eh_frame				
04	.ctors .dtors .jcr .data.rel.ro .dynamic .got .got.plt .data .bss				
05	.dynamic				
06	.note.ABI-tag .note.gnu.build-id				
07	.eh_frame_hdr				

图3-4 二进制可执行文件（/bin/ls）的Program头信息与Section/Segment结构映射信息

ELF 二进制文件的基本组成格式如图 3-5 所示。从总体上看，在 ELF 标准中定义的一系列头信息（ELF 头与 Program 头）以结构化形式描述了 ELF 二进制文件内各类资源（代码、数据及符号表等）的地址、系统权限及相关属性信息，利用这些信息则可以帮助链接器和编译器对应用程序的代码进行优化，以及在内存中分配应用程序运行前需要的各类资源，甚至在应用程序运行时为其进行动态链接的过程。

我们之前提到过，在大部分类 UNIX 系统中，用于进行动态链接的共享库，以及可以被直接运行的标准二进制可执行文件，均是以 ELF 标准为基础构建出来的。相比于普通应用程序而言，共享库并没有在其内部的 ELF 头中指定用于执行的入口函数地址，同时某些特殊的标志位（ET_EXEC、ET_DYN）也被设置为共享库专用的模式。而对于整个 ELF 二进制文件的组成结

构来说，两种文件格式并没有任何本质上的差别。下面我们将通过一个实例来感受一下在 Linux 系统上为传统二进制应用程序进行动态链接的具体过程。

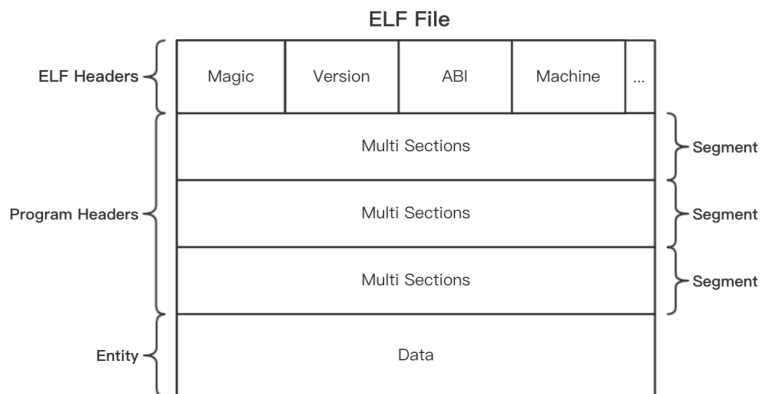


图3-5 ELF二进制文件的基本组成格式

构建共享库

首先编写主应用程序的源代码。主应用程序会在其源代码内部调用位于共享库中的外部共享变量，并根据该变量的具体值向控制台打印不同的计算结果。主应用程序的源代码如下。

```
program.c
#include <stdio.h>
// 声明需要进行外部引用的变量，这些变量将由共享库提供
extern int x;
extern int i[4];
// 主函数将作为可执行文件的入口函数
void main() {
    int *b = i + x;
    int x_position_val = *b;
    // 打印共享库中对应数组索引位置的数值
    printf("The result number is: %d\n", x_position_val);
    // 打印共享符号的实际引用地址
    printf("The address of symbol 'x' is: %p\n", &x);
    printf("The address of symbol 'i' is: %p\n", i);
    printf("The address of symbol 'printf' is: %p\n", printf);
}
```

接下来编写共享库的源代码。与主应用程序不同的是，在共享库中不需要编写程序的入口函数，即对应 C/C++ 语言中的“main”函数。共享库的源代码如下。

```
share.c
```

```
int i[4] = {0, 1, 2, 3};  
int x = 2;
```

可以看到, 这里我们直接在全局环境中声明了两个在主应用程序中使用到的外部共享变量, 并对它们赋予了初值。接下来需要将共享库的源代码编译成一个以 “.so” 为后缀的 Linux 二进制共享库文件。该文件为一个标准的 ELF 文件, 稍后我们将会看到。下面通过如下命令对共享库的源代码进行编译。

```
gcc -nostdlib -shared -fpic -o share.so share.c
```

上述命令使用 GCC 编译器将共享库的源代码编译成了一个标准的 ELF 二进制共享库模块文件。其中参数 “-nostdlib” 表示在编译过程中不默认引入系统的 C/C++ 标准库; “-shared” 表示编译的目标文件类型为一个 Linux 共享库; “-fpic” 表示生成与位置无关但与平台相关的模块文件, 该类型文件占有较小的存储空间。命令执行完毕后, 编译器会在当前目录下生成一个名为 “share.so” 的共享库文件。如图 3-6 所示, 通过 readelf 命令可以看到这是一个 DYN 类型的文件, 对应于 ELF 头中的 ET_DYN 字段, 该字段用于表示一个标准的共享库文件。

```
[root@iZ23ki8vt8tZ dynamic_linking]# readelf --segments ./share.so  
  
Elf file type is DYN (Shared object file)  
Entry point 0x280  
There are 5 program headers, starting at offset 64  
  
Program Headers:
```

图3-6 上述共享库的ELF头信息

最后, 需要对主应用程序源代码进行编译, 并在主应用程序运行时让其与共享库进行动态链接。我们通过如下命令对主应用程序的源代码进行编译。

```
gcc -fpic -o ./program ./share.so program.c
```

可以看到, 在编译主应用程序的源代码时还指定了需要进行动态链接的共享库位置, 并将其作为一个编译源参数传递给 GCC 编译器。命令执行完毕后, 编译器会在当前目录下生成名为 “program” 的二进制可执行文件。在保证相关联共享库所在位置不变的情况下, 我们可以直接在本地运行该二进制可执行文件。如图 3-7 所示为主应用程序的运行结果。这里应用程序直接打印出在共享库中定义的数组 “i” 在对应索引位置 “x” 处的数字值, 同时还打印出了符号 “x”、“i” 以及 “printf” 的具体引用地址。

```
[root@iZ23ki8vt8tZ dynamic_linking]# ./program  
The result number is: 2  
The address of symbol 'x' is: f171d360  
The address of symbol 'i' is: f171d350  
The address of symbol 'printf' is: f11d7f60
```

图3-7 上述主应用程序的运行结果

至此，我们已经了解了在 Linux 系统中对传统 ELF 二进制可执行文件进行动态链接操作的大致流程。这里并没有在任何文件编译或执行相关命令的过程时指定需要进行动态链接的参数或标志位。因此可以预见的是，整个 ELF 文件的动态链接过程确实是发生在应用程序的运行时状态。并且可以测试，当将上述例子中主应用程序使用到的共享库文件的所在位置变更后（比如删除或移动到其他目录中），主应用程序便无法再正常运行。接下来，我们将深入到主应用程序和共享库的内部 ELF 结构，来探索传统应用进行动态链接时的细节信息。

3.1.2 符号重定向（Symbol Relocation）

首先，我们通过 `readelf` 或 `ldd` 命令来查看主应用程序在运行时需要依赖的外部共享库。如图 3-8 所示，可以看到主应用程序在运行时需要依赖两个外部共享库，其名称分别为 `share.so` 和 `libc.so.6`。其中 `share.so` 是我们编写的专门用于为主应用程序提供外部共享变量的共享库；`libc.so.6` 是本地通用的 C 标准函数共享库，通过该共享库，我们可以在应用程序中使用诸如 `printf`、`malloc` 等 C 标准库中的函数。

```
[root@iZ23ki8vt8tZ dynamic_linking]# readelf --dynamic ./program

Dynamic section at offset 0x890 contains 25 entries:
   Tag              Type              Name/Value
0x0000000000000001 (NEEDED)           Shared library: [./share.so]
0x0000000000000001 (NEEDED)           Shared library: [libc.so.6]
0x000000000000000c (INIT)             0x400490
0x000000000000000d (FINI)             0x40071c
0x0000000000000019 (INIT_ARRAY)        0x600878
0x000000000000001b (INIT_ARRAYSZ)      8 (bytes)
```

图3-8 查看主应用程序使用的外部依赖库

之所以将应用程序使用共享库中代码的过程称为动态链接的过程，是因为应用程序在其运行时状态中是动态调用位于共享库中的代码的。为了能够正确地对应用程序源代码进行静态编译（AOT），便需要在源代码中事先声明将要在共享库中进行调用的符号具体类型，比如调用函数的函数签名或变量的具体值类型等。这样编译器便可以在主应用程序源代码的编译过程中为这些“共享符号”提前计算出其将要占用的内存空间大小，并完成各类相关资源的分配，同时也保证了程序的上下文逻辑具有正确的类型一致性。

当应用程序开始运行时，其内部需要被替换的共享符号则会被映射到当前的应用程序虚拟内存（PM）中，此时 ELF 二进制可执行文件在其内部指定的目标“动态链接器（Dynamic Linker，DL）”，便会通过一种名为“Relocation”的操作对这些共享符号进行引用地址的替换。比如对于上述主应用程序，我们可以通过 `readelf` 或 `nm` 等命令来查看编译器在其内部生成的符号表信息。在该表结构中存放着所有应用程序在运行过程中需要用到的独立数据元素，每一个数据元

素都指向一个独立的数据实体，比如一个变量或函数结构等。符号表中的每一个符号都具有全局唯一的名称，还记得如果尝试在一个 C++ 应用中使用函数的重载特性，编译器则会通过 Name Mangling 机制将源代码中所有重载函数的符号名唯一化。关于这部分内容我们曾在第 1 章中介绍过，这里不再赘述。

符号表

在标准的 ELF 二进制文件中，符号表有两种类型。其中一种类型为“.symtab”符号表，在该符号表中存放着编译器生成的 ELF 二进制文件内的所有符号信息，包括需要在程序运行过程中进行动态引用替换的共享符号，以及不需要进行动态替换的普通静态符号；另一种类型为“.symtab”符号表的子集，在该符号表中仅存放了需要在应用程序运行时状态下进行动态引用替换的共享符号，因此它又被称为“.dynsym”符号表。

当应用程序开始运行时，操作系统会将应用程序对应 ELF 二进制文件内.dynsym 符号表中的所有符号信息全部加载到内存中，以辅助应用程序完成后续的动态链接过程。由于该二进制文件内包含的是应用程序被编译完成后生成的与平台相关的二进制机器码，因此位于.symtab 符号表中的静态符号实际上并不会参与到应用程序的具体运行过程，即这些静态符号是否存在并不会影响应用程序的运行流程。通常来说，我们可以通过 strip 等命令将一个可执行 ELF 二进制文件中的.symtab 符号表信息完全移除，只保留存储有共享符号信息的.dynsym 符号表。这样做的好处是，可以在一定程度上减小应用程序文件占用的存储空间。但是由于缺少了用于描述程序逻辑和运行细节的完整符号信息，我们便无法通过各类 ELF 调试器来对该应用程序二进制文件进行完整分析。如图 3-9 所示为上一节示例中编译器在主应用程序内部生成的符号表信息。

```
[root@i223ki8vt8tZ dynamic_linking]# readelf --symbols ./program

Symbol table '.dynsym' contains 9 entries:
  Num:   Value              Size Type      Bind   Vis      Ndx Name
   0: 0000000000000000      0 NOTYPE    LOCAL DEFAULT UND
   1: 0000000000000000      0 FUNC      GLOBAL DEFAULT UND printf@GLIBC_2.2.5 (2)
   2: 0000000000000000      0 NOTYPE    WEAK  DEFAULT UND __gmon_start__
   3: 0000000000000000      0 FUNC      GLOBAL DEFAULT UND __libc_start_main@GLIBC_2.2.5 (2)
   4: 0000000000000000      0 OBJECT    GLOBAL DEFAULT UND x
   5: 0000000000000000      0 OBJECT    GLOBAL DEFAULT UND i
   6: 00000000000000ac0      0 NOTYPE    GLOBAL DEFAULT ABS _end
   7: 00000000000000abc      0 NOTYPE    GLOBAL DEFAULT ABS _edata
   8: 00000000000000abc      0 NOTYPE    GLOBAL DEFAULT ABS __bss_start

Symbol table '.symtab' contains 67 entries:
  Num:   Value              Size Type      Bind   Vis      Ndx Name
   0: 0000000000000000      0 NOTYPE    LOCAL DEFAULT UND
   1: 0000000000400200      0 SECTION   LOCAL DEFAULT 1
   2: 000000000040021c      0 SECTION   LOCAL DEFAULT 2
   3: 000000000040023c      0 SECTION   LOCAL DEFAULT 3
   4: 0000000000400260      0 SECTION   LOCAL DEFAULT 4
```

图3-9 EFL二进制文件内的符号表信息

可以看到，图中第一部分内容展示了该应用程序 ELF 二进制文件内 `.dynsym` 符号表中各符号的描述信息，我们在主应用程序的源代码中通过 `extern` 关键字声明的、需要通过动态链接过程从共享库中获取的两个变量“`x`”和“`i`”均被定义在此处。除此之外，我们在源代码中用到的同样需要从 C 标准库中获取的 `printf` 函数也被定义在该表中。总的来说，这两种类型的符号表均有其各自的具体使用场景。首先在 `.dynsym` 符号表中存放着需要在应用程序运行过程中通过动态链接来获取的所有符号信息，这些符号信息确保了应用程序的正常运行流程，因此不能从 ELF 二进制文件中移除。而定义在 `.symtab` 符号表中的部分静态符号信息仅用于应用程序的编译流程及调试环节，其是否存在，对于已经被编译成平台相关机器码的二进制应用程序文件本身来说没有任何影响，因此可以从应用程序对应二进制文件中安全地移除该符号表。

重定向 (Relocation)

从上文中我们已经了解到，对于那些需要在应用程序运行过程中通过动态链接过程才能被使用的符号，应用程序会在之前的静态编译过程中根据这些符号的具体类型及签名等信息，为它们生成一个默认的符号引用地址并保留适当的资源和环境上下文。当应用程序运行时，其使用到的共享库会连同应用程序本身一起被加载到物理内存中，每个独立的应用程序进程都会在其各自的 PM 空间中进行以“段”为结构的资源分配。应用程序内特定的动态链接器会根据存放在共享库中符号的实体引用地址，来修改此前应用程序 PM 空间中对应该符号的默认引用地址。当应用程序内所有动态符号的引用地址都被修改为指向共享库中对应符号的实体引用地址后，应用程序便可以开始正常运行了。

对于这种动态链接器在应用程序运行状态下即时修改 PM 空间中动态符号引用地址的过程，我们称之为 `Relocation` 操作，翻译成中文即“重定向”操作，即对符号实体的实际引用地址进行重定向。接下来，我们通过如图 3-10 所示的命令来查看编译器在之前的主应用程序内部标记出的需要进行引用地址重定向的符号信息。

```
[root@i223ki8vt8tZ dynamic_linking]# readelf -r ./program

Relocation section '.rela.dyn' at offset 0x400 contains 4 entries:
  Offset             Info                Type             Sym. Value      Sym. Name + Addend
000000600a70 000100000006 R_X86_64_GLOB_DAT 0000000000000000 printf + 0
000000600a78 000200000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0
000000600a80 000400000006 R_X86_64_GLOB_DAT 0000000000000000 x + 0
000000600a88 000500000006 R_X86_64_GLOB_DAT 0000000000000000 i + 0

Relocation section '.rela.plt' at offset 0x460 contains 2 entries:
  Offset             Info                Type             Sym. Value      Sym. Name + Addend
000000600aa8 000100000007 R_X86_64_JUMP_SLO 0000000000000000 printf + 0
000000600ab0 000300000007 R_X86_64_JUMP_SLO 0000000000000000 __libc_start_main + 0
```

图3-10 上述主应用程序的符号重定向信息

可以看到，在一个标准的 ELF 二进制文件中，需要重定向的符号分为两种类型。第一种是

位于名为“.rela.dyn”的重定向专用 Section 结构中的符号,在该 Section 中存放的共享符号主要用于对某个变量的具体值或所在地址等符号相关信息进行引用(比如这里的符号“x”主要用于对变量“x”的具体值进行引用,符号“printf”主要用于对该函数名变量的所在地址进行引用,但不涉及函数体本身);第二种是位于名为“.rela.plt”的重定向专用 Section 结构中的符号,在这个 Section 结构中存放的共享符号主要用于对源代码中某个共享函数的调用过程进行函数体实际位置的引用。总的来说,存放在.rela.dyn 结构中的符号主要用于对变量进行引用,而位于.rela.plt 结构中的符号则主要用于对函数体进行引用。

在重定向列表中,每一条记录都对应于一个共享符号的重定向信息。其中,Offset 字段指明该共享符号引用地址被存放在 PM 空间中的具体偏移位置,比如从图 3-10 中可以看到,共享符号“x”的实际引用地址被存放在进程 PM 空间中相对基本地址偏移量为“0x600a80”的位置上。Type 字段指明重定向符号的具体类型(比如这里的“R_X86_64_GLOB_DAT”指明该重定向符号是全局变量类型),该字段的具体内容与处理器和操作系统架构的具体应用程序二进制接口(Application Binary Interface, ABI)标准定义有关。最后一个字段指明该重定向符号的具体名称,以及在进行符号引用地址修正时需要增加的“偏移量加数(Addend)”。偏移量加数一般会被使用在对数组关联变量的引用地址进行修正的过程中。比如对于如下 C/C++代码。

```
extern int i[4];
int *j = i + 2;
```

当将上述代码编译成 ELF 二进制文件时,在该文件内的.rela.dyn 重定向列表中将会存在两个重定向符号。第一个为数组变量“i”产生的符号“i+0”;第二个为整型指针“j”产生的符号“i+8”。这里的“i+8”表示在修正指针“j”对应的符号引用地址时,需要在变量“i”的真实数据引用地址基础上进行加 8 操作。其中的加数“8”是由于指针“j”所指向的地址,在数组首地址变量“i”的地址基础上,向内存高位方向移动了 8 字节(对应两个 4 字节的 int 类型)的地址偏移量而得来的。

Load-time Relocs

通常来说,在编译一个共享库时,可以选择是否需要在编译命令中添加“-fPIC”或“-fpic”选项。这两个选项的主要用处是,让主应用程序能够以 PIC 的方式进行共享库符号的重定向过程。在默认情况下,主应用程序会以 Load-time Relocs 这种方式来进行符号重定向。

所谓的 Load-time Relocs 就是指载入时重定向。顾名思义,这种共享符号的重定向方式是指当共享库被加载到内存中时,动态链接器便立即开始对主应用程序使用到的符号进行重定向操作。接下来,我们将通过一个具体的实例来了解载入时重定向的具体工作流程及其优缺点。这里

将具体引用的共享符号分为两种：一是被定义在共享库源代码内，并且直接在当前共享库内部的其他地方引用的变量符号；二是被定义在共享库源代码内，同时又被其他外部应用程序引用的变量符号。下面以第一种引用为例进行介绍。首先需要将如下 C/C++ 源代码编译成一个共享库。

```
share.c
int x = 11;

int func(int a, int b) {
    x += a;
    return x + b;
}
```

可以看到，在该共享库的源代码中，我们在全局环境中定义了一个名为“x”的变量，并对其进行了初始化操作。在接下来定义的全局 func 函数中，则引用了全局变量“x”，并返回了经过一系列数学计算后的结果。下面我们在本地命令行中通过 gcc 命令来编译这段 C/C++ 源代码。

```
gcc -nostdlib -shared -m32 -o share.so share.c
```

在上述命令中，指定了几个必要的参数。“-nostdlib”参数告知编译器不需要在编译过程中链接默认的 C/C++ 标准库；“-shared”参数指定编译器生成的目标文件类型为一个共享库；“-m32”参数设置编译器以基于 x86 架构的模式来构建目标文件。需要注意的是，由于载入时重定向本身的历史问题，导致该重定向方式并不能在基于 x64 架构的模式下使用。最后一个参数“-o”指定目标文件的文件名和具体类型。当该条命令执行结束后，编译器会在当前目录下生成一个名为“share.so”的共享库文件。

现在我们通过 readelf 命令来查看该共享库文件的入口地址。如图 3-11 所示，主应用程序在调用共享库时会从其 VAS 相对地址 0x204 处将它的源代码加载到应用程序内部的虚拟地址空间中运行。注：这个示例使用的操作系统环境虽然与之前的实践环境不同，但并不会对最终结果产生影响。

```
[root@iZbp173padvrcc9zlkj9vkZ load_time_relocs]# readelf -h share.so
ELF Header:
  Magic:   7f 45 4c 46 01 01 00 00 00 00 00 00 00 00 00 00
  Class:                                ELF32
  Data:                                      2's complement, little endian
  Version:                               1 (current)
  OS/ABI:                                UNIX - System V
  ABI Version:                           0
  Type:                                    DYN (Shared object file)
  Machine:                                Intel 80386
  Version:                                0x1
  Entry point address:                    0x204
  Start of program headers:              52 (bytes into file)
```

图3-11 共享库文件的入口地址信息

如果通过相关命令来查看该共享库内部的所有 Section 信息,则会发现该模块的入口地址正对应着共享库“.text”段的起始地址,在该段结构中存放着所有共享库模块内部的源代码信息。接下来,我们仍然通过 readelf 命令并附加“--relocs”(同-r)参数来查看该共享库模块内部的所有符号重定向信息,即被重定向的符号信息,以及对重定向符号进行修正的偏移地址。相关命令的执行过程及输出结果如图 3-12 所示。

```
[root@iZbp173padvrcc9zlkj9vkZ load_time_relocs]# readelf --relocs share.so

Relocation section '.rel.dyn' at offset 0x1ec contains 3 entries:
  Offset      Info    Type           Sym.Value    Sym. Name
00000209     00000101 R_386_32      00002000     x
00000213     00000101 R_386_32      00002000     x
00000219     00000101 R_386_32      00002000     x
```

图3-12 共享库文件的符号重定向信息

从输出结果中可以看到,当主应用程序从共享库的特定地址段开始加载时,动态链接器需要在共享库的虚拟地址空间偏移量分别为 0x209、0x213 和 0x219 的三个位置对符号“x”的引用地址进行修正。通过为 readelf 命令附加--sections 参数可以看到,以上三个符号重定向过程均发生在共享库的“.text”段结构中。如图 3-13 所示,存放源代码的“.text”段结构起始于偏移地址 0x204 处,并且该 Section 占用的总长度 0x20 正好将上述重定向的发生位置囊括其中。

```
[root@iZbp173padvrcc9zlkj9vkZ load_time_relocs]# readelf --sections share.so
There are 15 section headers, starting at offset 0x1250:

Section Headers:
[Nr] Name              Type              Addr             Off             Size            ES Flg Lk Inf Al
[ 0]                     NULL              00000000          000000          000000 00      0 0 0
[ 1] .note.gnu.build-id    NOTE              00000114          000114          000024 00      A 0 0 4
[ 2] .gnu.hash             GNU_HASH          00000138          000138          000034 04      A 3 0 4
[ 3] .dynsym               DYNSYM            0000016c          00016c          000060 10      A 4 1 4
[ 4] .dynstr               STRTAB            000001cc          0001cc          000020 00      A 0 0 1
[ 5] .rel.dyn              REL               000001ec          0001ec          000018 08      A 3 0 4
[ 6] .text                 PROGBITS          00000204          000204          000020 00     AX 0 0 1
[ 7] .eh_frame_hdr         PROGBITS          00000224          000224          000014 00      A 0 0 4
[ 8] .eh_frame             PROGBITS          00000238          000238          000038 00      A 0 0 4
[ 9] .dynamic              DYNAMIC           00001f88          000f88          000078 08      WA 4 0 4
[10] .data                 PROGBITS          00002000          001000          000004 00      WA 0 0 4
[11] .comment              PROGBITS          00000000          001004          00002d 01      MS 0 0 1
[12] .shstrtab             STRTAB            00000000          001031          000087 00      0 0 1
[13] .symtab               SYMTAB            00000000          0010b8          000150 10      14 16 4
[14] .strtab               STRTAB            00000000          001208          000047 00      0 0 1
```

图3-13 共享库文件的Section信息

下面我们将通过 objdump 命令深入到共享库的反编译汇编代码中一探究竟。通过如下命令可以对共享库的二进制文件进行反汇编处理。

```
objdump -d -Intel share.so
```

我们为 objdump 命令指定了两个参数:“-d”参数表示对指定的目标二进制文件进行反编译

处理，生成该 ELF 二进制文件所对应的汇编代码；参数“-Mintel”（它是一个复合参数，其中“-M”为参数名，“intel”为该参数的具体参数值）指定反编译器需要参考的处理器指令集类型，这里指定反编译器按照 Intel 处理器架构的相关指令集对目标文件进行反编译处理。该命令的执行结果如图 3-14 所示。

```
[root@izbp173padvrcc9zlkj9vkZ load_time_relocs]# objdump -d -Mintel share.so

share.so:      file format elf32-i386

Disassembly of section .text:

00000204 <func>:
204:  55                push    ebp
205:  89 e5             mov     ebp,esp
207:  8b 15 00 00 00 00 mov     edx,DWORD PTR ds:0x0
20d:  8b 45 08          mov     eax,DWORD PTR [ebp+0x8]
210:  01 d0             add     eax,edx
212:  a3 00 00 00 00   mov     ds:0x0,eax
217:  8b 15 00 00 00 00 mov     edx,DWORD PTR ds:0x0
21d:  8b 45 0c          mov     eax,DWORD PTR [ebp+0xc]
220:  01 d0             add     eax,edx
222:  5d                pop     ebp
223:  c3                ret
```

图3-14 共享库文件的反编译汇编代码

在图 3-14 所示的汇编代码中，首先按照第一列给出的偏移地址，找到前面三个符号重定向过程发生的位置。这里将以在 0x213 偏移位置处发生的重定向过程为例进行介绍。可以看到，该位置处的符号重定向过程发生在地址“0x212”这一行对应的汇编指令中。这行指令的主要目的是，通过 mov 指令将位于共享库虚拟地址段偏移位置“0x0”处的值存放到寄存器 eax 中。实际上，这里的“0x0”便是编译器为变量“x”设置的默认偏移位置，即需要通过重定向过程进行修正的值。另外两个符号重定向过程发生的位置对应的汇编代码与之类似，这里不再赘述。

接下来，我们将编写一个主应用程序，并调用定义在共享库中的全局函数。该应用程序的 C/C++源代码如下。

```
program.c
#define _GNU_SOURCE
#include <link.h>
#include <stdlib.h>
#include <stdio.h>
// dl_iterate_phdr 函数的处理钩子函数
static int handler (struct dl_phdr_info* info, size_t size, void* data) {
    printf("name=%s (%d segments) address=%p\n",
           info->dlpi_name, info->dlpi_phnum, (void*)info->dlpi_addr);
    for (int j = 0; j < info->dlpi_phnum; j++) {
```

```

        printf("\t\t header %2d: address=%10p\n", j,
            (void*) (info->dlpi_addr + info->dlpi_phdr[j].p_vaddr));
        printf("\t\t\t type=%u, flags=0x%X\n",
            info->dlpi_phdr[j].p_type, info->dlpi_phdr[j].p_flags);
    }
    printf("\n");
    return 0;
}
// 声明共享库中函数的原型
extern int func(int, int);
// 主函数
int main (int argc, char* argv[]) {
    // 通过 Linux 下的 dl_iterate_phdr 库函数来查看共享库的运行时信息
    dl_iterate_phdr(handler, NULL);
    // 调用共享库中的全局函数
    int t = func(1, 2);
    return t;
}

```

在上面的主应用程序源代码中，我们在主函数内调用了上述共享库向外部暴露出的全局函数，同时还通过调用 Linux 系统下专有的 `dl_iterate_phdr` 库函数来打印主应用程序在运行过程中依赖的各个共享库的加载情况。`dl_iterate_phdr` 函数的第一个参数是一个钩子函数，当应用程序成功加载共享库时，该函数会立即回调钩子函数并向其传递一个含有共享库相关信息的结构体。

我们通过如下命令对上面这段 C/C++ 源代码进行编译。需要注意的是，共享库文件与主应用程序的源代码文件需要被放到同一个编译目录下，才能够进行正确编译。

```
gcc -std=c99 -g -m32 -o ./program ./share.so program.c
```

上面命令中的“-g”参数主要用来让编译器生成调试信息，以便后面进行调试。当主应用程序被编译完成后，我们直接使用 GDB（类 UNIX 系统中的可执行二进制文件调试器）对该主应用程序进行运行时调试。首先通过下面命令以安静模式将主应用程序加载到 GDB 调试器中。

```
gdb -q ./program
```

上面命令中的参数“-q”表示让调试器以安静模式来加载应用程序。在安静模式下，调试器不会向控制台输出诸如版权、版本号、基本介绍等相关基本信息。当命令执行完成后，便直接进入了 GDB 的命令行控制台中，在这里我们可以直接对应用程序的执行进程进行控制和调试，如图 3-15 所示。

```
[root@iZbp173padvrcc9zlkj9vkZ load_time_relocs]# gdb -q program
Reading symbols from /root/workspace/load_time_relocs/program...done.
(gdb)
(gdb)
```

图3-15 使用GDB调试共享库文件

应用程序加载完成后，我们可以在其主代码文件中设置断点。当应用程序在 GDB 调试器中运行时，将在该断点位置暂停进程。设置断点的命令如下（这里的命令前缀“(gdb)”表示当前的命令执行环境为 GDB 调试器内部的交互控制台）。

```
(gdb) b program.c:26
```

通过该命令，我们在应用程序主代码文件 `program.c` 的第 26 行（主函数的最后位置）设置了一个断点。该命令的运行结果如图 3-16 所示。

```
(gdb) b program.c:26
Breakpoint 1 at 0x804863e: file program.c, line 26.
```

图3-16 在GDB调试环境中为共享库源代码设置断点

断点设置完毕后，便可以让应用程序运行起来。命令如下。

```
(gdb) r
```

在应用程序运行时向控制台输出的众多信息中，我们找到了如图 3-17 所示的内容。这段内容便是 `dl_iterate_phdr` 函数返回的当 `share.so` 动态库被加载到主应用程序虚拟地址空间时的一些具体情况。可以看到，整个共享库被加载到主应用程序虚拟地址空间的偏移位置“`0xf7fd6000`”处，同时该位置也正对应着共享库第一个 Segment 结构的起始地址。根据共享库中各 Segment 结构的大小及其相对偏移量，我们知道图中各个 header 所在的偏移位置正好按照顺序分别对应着共享库中各个 Segment 结构的起始地址（比如共享库中第二个类型为 `LOAD` 的 Segment 结构距离共享库的起始地址偏移量为“`0x1f88`”，而这正好对应于图中第二个 header 结构所在的地址“`0xf7fd6000 + 0x1f88`”，即“`0xf7fd7f88`”）。

```
type=1685382480, flags=0x4
name=./share.so (7 segments) address=0xf7fd6000
  header 0: address=0xf7fd6000
    type=1, flags=0x5
  header 1: address=0xf7fd7f88
    type=1, flags=0x6
  header 2: address=0xf7fd7f88
    type=2, flags=0x6
  header 3: address=0xf7fd6114
    type=4, flags=0x4
  header 4: address=0xf7fd6224
    type=1685382480, flags=0x4
  header 5: address=0xf7fd6000
    type=1685382481, flags=0x6
  header 6: address=0xf7fd7f88
    type=1685382482, flags=0x4
name=/lib/libc.so.6 (10 segments) address=0xf7e10000
```

图3-17 GDB调试的输出信息

同样的,通过如下 GDB 命令,我们可以查看在共享库中被加载到数据段(.data)的变量“x”其所在的地址,如图 3-18 所示。

```
(gdb) p &x
```

```
(gdb) p &x
$1 = (<data variable, no debug info> *) 0xf7fd8000 <x>
```

图3-18 通过GDB查看变量“x”的所在地址

从图 3-18 中可以看到,变量“x”所在的实际偏移地址也正好与我们通过 nm 命令得到的共享库中符号的偏移地址相吻合。这里的“0xf7fd8000”是由共享库的加载基地址“0xf7fd6000”加上变量“x”对应符号的存储偏移地址“0x2000”得来的。

由于我们之前在 GDB 调试器的交互控制台中为主应用程序源代码的执行流程设置了一个断点,因此当前应用程序并没有从内存中退出。由于程序已经开始运行,因此动态链接器也已经完成了对共享库进行的载入时重定向过程。下面我们可以通过如下命令直接查看位于当前共享库“.text”段结构中 func 函数所对应的汇编代码。

```
disas /r func
```

还记得我们之前对共享库 share.so 进行静态反编译后生成的汇编代码吗?在载入时重定向发生之前,位于代码段(.text)偏移位置 0x213 处的符号“x”其重定向地址被默认设置为“0x0”。对应的,当共享库被加载到主应用程序的虚拟地址空间时,该位置对应的实际符号地址会被动态链接器进行改写。我们在偏移地址(0x213)的基础上累加共享库的载入基地址(0xf7fd6000),便可以得到该符号重定向的实际地址(0xf7fd6213)。从图 3-19 中可以看到,该处符号重定向的对应值已经由原先的“0x0”被改写为“0xf7fd8000”,即符号“x”在共享库数据段(.data)中的实际存放位置。

```
(gdb) disas /r func
Dump of assembler code for function func:
0xf7fd6204 <+0>: 55      push    %ebp
0xf7fd6205 <+1>: 89 e5   mov     %esp,%ebp
0xf7fd6207 <+3>: 8b 15 00 80 fd f7   mov     0xf7fd8000,%edx
0xf7fd620d <+9>: 8b 45 08   mov     0x8(%ebp),%eax
0xf7fd6210 <+12>: 01 d0   add     %edx,%eax
0xf7fd6212 <+14>: a3 00 80 fd f7   mov     %eax,0xf7fd8000
0xf7fd6217 <+19>: 8b 15 00 80 fd f7   mov     0xf7fd8000,%edx
0xf7fd621d <+25>: 8b 45 0c   mov     0xc(%ebp),%eax
0xf7fd6220 <+28>: 01 d0   add     %edx,%eax
0xf7fd6222 <+30>: 5d      pop     %ebp
0xf7fd6223 <+31>: c3      ret
End of assembler dump.
```

图3-19 函数func对应的汇编代码

以上介绍的便是载入时重定向的一个实际例子。接下来我们将从多个角度来总结该重定向方式存在的几个问题。

首先，毫无疑问的是，载入时重定向会显著增加应用程序的启动时间。这个问题在一些大型的多共享库依赖应用程序中会十分明显。其次，由于载入时重定向会动态地修改共享库“.text”段中的指令代码（如重定向的偏移地址），这将导致同时被映射到其他主应用程序虚拟地址空间中对应该共享库实例的特定段代码也会被修改。但实际上，共享库在各个应用程序虚拟地址空间中的具体加载位置是无法提前预知的。因此对每一个应用程序来说，在共享库中各重定向符号的实际修正值也都并不相同。为了解决这个问题，动态链接器会为每一个处于不同应用程序VAS加载位置的共享库，在内存中都分配一个独立且完整的共享库实例副本。虽然这个办法可以解决上述问题，但同时也使共享库失去了其可以被“共享”的特性。不仅如此，一个可写的“.text”段结构也使得应用程序内部暴露出了一定的安全隐患（如可篡改代码），这也极大增加了应用程序被黑客利用的风险。

在之前的实例中，我们仅介绍了第一种共享符号引用方式（同时在共享库中定义和使用共享符号）的载入时重定向细节及其存在的问题。接下来我们介绍第二种共享符号引用方式（在共享库中定义共享符号，在主应用程序中使用该共享符号），并探究其与第一种引用方式有哪些不同。这里我们不需要修改共享库的源代码，新的主应用程序代码如下。

```
program_second.c
#include <stdio.h>
// 声明需要使用到的外部共享库符号原型
extern int func(int, int);
extern int x;

int main (int argc, char* argv[]) {
    printf("The address of symbol 'x': %p\n", &x);
    int _t = func(1, 2);
    return _t;
}
```

通过使用相同的编译指令，我们将上述 C/C++源代码编译成对应的二进制文件。接下来直接使用 `readelf` 命令来观察该二进制文件内部的符号重定向信息。如图 3-20 所示，这里只截取了变量符号引用部分的重定向信息。

```
[root@iZbp173padvrcc9zlkj9vkZ load_time_relocs]# readelf -r program_second

Relocation section '.rel.dyn' at offset 0x32c contains 2 entries:
   Offset      Info    Type           Sym.Value   Sym. Name
08049ffc  00000206  R_386_GLOB_DAT  00000000    __gmon_start__
0804a020  00000605  R_386_COPY      0804a020    x
```

图3-20 新主应用程序的符号重定向信息

可以看到, 变量“x”的符号重定向地址为“0x804a020”, 如果与主应用程序各 Section 结构的偏移地址信息进行对照, 则会发现该重定向地址正好位于主应用程序的“.bss”段结构, 即应用程序的未初始化数据段中。该符号重定向的类型标识“R_386_COPY”告知动态链接器在修正时需要进行的操作, 即直接将共享库中变量“x”的具体值拷贝到该符号重定向的偏移位置处。如果按照我们之前对共享库的分析流程, 直接使用 GDB 分析主应用程序 main 函数中的反编译汇编代码, 会发现应用程序在其运行时代码中, 直接使用了其自身虚拟地址空间偏移位置“0x804a020”处存放的值来作为变量“x”的实际值。这种看上去简单、粗暴的重定向及修正方式, 会引来另一个关于载入时重定向的性能问题, 即每一个使用了共享库中全局变量的应用程序, 都会将对应符号的变量值直接拷贝到当前应用进程的虚拟地址空间中, 而不是在各个应用进程中进行共享。这同样也违背了共享库本身最重要的共享特性。

为了解决载入时重定向的一系列问题, 在大部分基于 64 位处理器架构的操作系统中, 我们开始尝试将共享库编译为 PIC 结构, 进而让动态链接器以一种间接的方式来处理各类共享符号的重定向过程。

PIC

对于 PIC (Position Independent Code, 位置无关代码), 我们可以直接从字面意义上来进行理解, 即代码所存放的具体位置并不会影响其本身的正常执行流程。而对于一个基于 PIC 方式编译的共享库来说, 便意味着无论该共享库被加载到应用程序虚拟地址空间中的哪个地方, 应用进程都可以正常地使用该共享库对外提供的任何资源和功能。

整个 PIC 共享库的实现方式非常简单。共享库作为一个标准的 ELF 二进制文件, 在代码编译阶段后进行链接过程时, 其内部各个 Section 结构的相对偏移位置便已经确定。如图 3-21 所示, 这里假设共享库代码段(.text)中的某行指令想要引用位于其数据段(.data)中的某个变量的具体值。当共享库处于链接阶段时, 链接器已经可以确定的是从该行指令到其数据段的实际偏移地址。因此需要一种办法, 可以在主应用程序运行之前就与共享库中代码段的指令提前知晓并确定目标符号所在的位置。这样, 不仅可以省去动态链接器在共享库载入时进行的符号重定向过程, 而且共享库的代码段(.text)也不需要再运行时进行任何修改。这种办法从某种程度上提高了共享库的加载和使用效率, 同时也降低了应用程序被利用的安全风险。

为了实现这种共享符号的引用方式, 我们只需要在共享库的数据段(.data)结构中设置一个用来映射各符号名与其实际偏移地址关系的表结构即可。在共享库被加载到主应用程序的虚拟地址空间之前, 其代码段(.text)中需要引用共享符号的相关指令并不会直接去引用对应符号的实际位置, 而是通过我们预先设置好的含有各符号名与其实际偏移地址对应关系的表结构

来间接地引用符号的具体值，这个表结构我们将其称为 GOT（全局偏移表）。

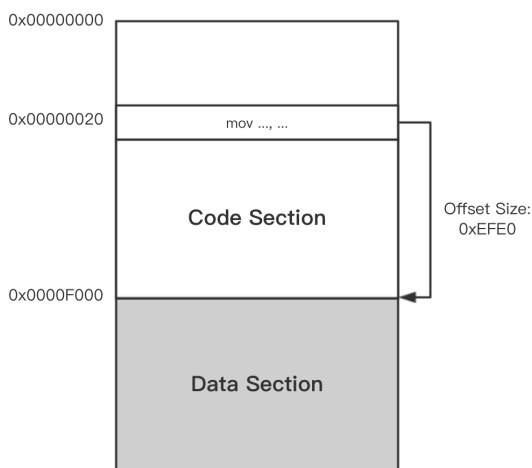


图3-21 代码段与数据段的交互逻辑关系

由于在共享库代码段中需要引用的各符号其具体名称在主应用程序运行前便已经确定，因此 GOT 结构中所有条目的符号名字段便可以事先被确定下来。GOT 中每一个符号与地址的对应关系条目都会被分配一个唯一的 GOT 偏移地址，而这些偏移地址便会被共享库代码段中的相关符号引用指令直接使用，即指令会通过这些偏移地址间接地引用符号的值。由于 GOT 的使用，共享符号重定向过程的发生位置从原来的共享库代码段（.text）转移到了 GOT 所在的数据段（.data）中。当共享库被加载到主应用程序的虚拟地址空间时，动态链接器便会根据当前共享库的载入基地址，来动态地修正 GOT 中各条目对应符号的实际引用地址（重定向）。

这种简单的间接符号地址引用方式使得动态链接器不必再去动态地修改共享库实例代码段（.text）中的代码，保障了共享库和应用程序的代码安全。同时，内存中的同一份共享库实例可以被映射到主应用程序虚拟地址空间中的任何位置，而不用担心其无法被正常使用，这使得共享库可以真正地在应用程序之间进行共享，即达到了“一份实例，多处使用”的目的。不仅如此，相比载入时重定向而言，基于 GOT 实现的 PIC 共享库使得应用程序的加载时间得以缩短（因为不需要动态链接器在应用加载时修改共享库实例的代码），符号的重定向过程不会再影响应用程序的正常运行。

在实际的类 UNIX 操作系统中，基于 PIC 进行的共享库动态链接过程不仅需要 GOT 的帮助，而且还需要 PLT 结构来为我们处理各类函数调用的重定向过程。实际上，在 GOT 中仅存放了对变量的值或地址等数据进行调用时产生的重定向信息，而对于代码中一系列函数调用过

程产生的重定向信息，则会由 GOT 与 PLT 共同负责处理和保存。GOT 与 PLT 的相互协作，使得 Linux 系统下传统应用程序与共享库进行动态链接的过程变得十分通用和稳定。

3.1.3 GOT (Global Offset Table, 全局偏移表)

本节我们将通过一个简单的例子来探索 GOT 在一个 ELF 二进制文件内的具体结构，以及在进行共享库动态链接过程中所扮演的角色。首先，我们需要将如下 C/C++ 源代码编译成一个与加载位置无关的 PIC 共享库。在这段代码中我们声明了一个全局变量 “x” 和一个全局函数 “func”，并且该全局函数还在其函数体内使用了全局变量 “x” 的值。

```
share.c
int x = 10;
int y = 11;

int func (int a, int b) {
    return a + b + x + y;
}
```

在本地命令行中，通过如下命令对源代码进行编译。

```
gcc -nostdlib -shared -fpic -o share.so share.c
```

需要注意的是，在上面的命令中一定要携带 “-fpic” 或 “-fPIC” 参数，这样才能够将目标共享库编译成与加载位置无关的 ELF 二进制文件。当共享库编译完成后，我们直接使用 readelf 命令来查看该共享库内部的符号重定向信息。该命令的执行结果如图 3-22 所示。

```
[root@iZ23ki8vt8tZ got]# readelf -r share.so
Relocation section '.rela.dyn' at offset 0x2e0 contains 2 entries:
   Offset             Info           Type           Sym. Value      Sym. Name + Addend
000000200468      000200000006  R_X86_64_GLOB_DAT 000000000200490  x + 0
000000200470      000600000006  R_X86_64_GLOB_DAT 000000000200494  y + 0
```

图3-22 共享库内部的符号重定向信息

可以看到，在该 PIC 共享库的二进制文件中，一共有两条符号重定向信息，其偏移地址分别为 “0x200468” 和 “0x200470”。如果将这两条重定向信息的地址与共享库中各个 Section 结构的偏移地址进行对照，则会发现它们的地址均位于共享库内一个名为 “.got” 的 Section 结构中。该共享库中各 Section 结构的相关信息如图 3-23 所示，这里只截取了其中的一部分。仔细查看，其中的 “.got” 结构起始于偏移地址 0x200468，并且其占用的长度为 0x10 字节，经过换算即 16 字节大小。

[9]	.dynamic	DYNAMIC	0000000000200388	00000388
	00000000000000e0	0000000000000010	WA 4 0 8	
[10]	.got	PROGBITS	0000000000200468	00000468
	0000000000000010	0000000000000008	WA 0 0 8	
[11]	.got.plt	PROGBITS	0000000000200478	00000478
	0000000000000018	0000000000000008	WA 0 0 8	
[12]	.data	PROGBITS	0000000000200490	00000490
	0000000000000008	0000000000000000	WA 0 0 4	

图3-23 GOT对应的Section结构的偏移位置及长度信息

其实，从该 Section 结构的名称就能猜出其扮演的具体角色和作用，即名为“.got”的 Section 结构正对应着我们之前介绍的共享库内的 GOT 结构，在这个表结构中存放着所有变量共享符号的名称与其实际值所在位置的对应关系。不仅如此，图 3-23 中名为“.got.plt”的 Section 结构也是整个 GOT 结构的一部分，只不过这部分 Section 结构主要用来与 PLT 结构进行协作，共同完成主应用程序内函数调用过程对应的符号重定向过程。

所有存放在 GOT 中的表项均由两部分信息组成，总共 8 字节大小。上面的重定向符号“x”与“y”在 GOT 中的具体存储方式如图 3-24 所示。整个表结构信息分为两列，第一列为符号名称，该名称主要用来为代码段中使用到该符号的相关指令做间接引用地址的查找索引。通过正确的符号名称，动态链接器可以为这些指令在 GOT 中查找到对应符号的正确引用地址。第二列则主要用来存放在应用程序运行时为符号分配的绝对偏移地址，该地址将会在主应用程序运行过程中被动态链接器重定向为正确的偏移地址。

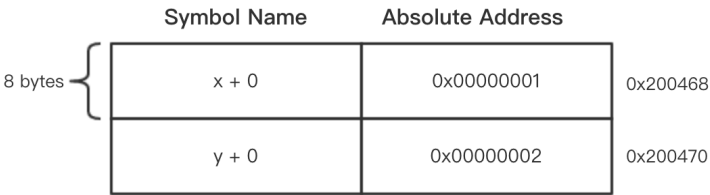


图3-24 重定向符号“x”与“y”在GOT中的具体存储方式

下面我们通过 objdump 命令来查看该共享库二进制文件内函数 func 的静态汇编代码，如图 3-25 所示。可以看到，在位于代码段(.text)内偏移地址“0x322”和“0x32d”处的两行汇编指令，分别引用了 GOT 中变量“x”和“y”所在的表项地址。当主应用程序开始运行时，共享库中的这段汇编代码便可以通过该表项地址间接地从 GOT 中得到符号的实际位置。

下面我们编写用于加载该共享库的主应用程序源代码，并通过 GDB 调试器来分析 GOT 中的表项数据在应用程序运行时的变化过程。

```

[root@iZ23ki8vt8tZ got]# objdump -d -Intel share.so

share.so:      file format elf64-x86-64

Disassembly of section .text:

0000000000000310 <func>:
310:  55                push    rbp
311:  48 89 e5          mov     rbp, rsp
314:  89 7d fc          mov     DWORD PTR [rbp-0x4], edi
317:  89 75 f8          mov     DWORD PTR [rbp-0x8], esi
31a:  8b 45 f8          mov     eax, DWORD PTR [rbp-0x8]
31d:  8b 55 fc          mov     edx, DWORD PTR [rbp-0x4]
320:  01 c2            add     edx, eax
322:  48 8b 05 3f 01 20 00 mov     rax, QWORD PTR [rip+0x20013f]          # 200468 <_DYNAMIC+0xe0>
329:  8b 00            mov     eax, DWORD PTR [rax]
32b:  01 c2            add     edx, eax
32d:  48 8b 05 3c 01 20 00 mov     rax, QWORD PTR [rip+0x20013c]          # 200470 <_DYNAMIC+0xe8>
334:  8b 00            mov     eax, DWORD PTR [rax]
336:  01 d0            add     eax, edx
338:  5d                pop     rbp
339:  c3                ret

```

图3-25 共享库二进制文件内函数func的静态汇编代码

主应用程序源代码如下。

```

program.c
extern int func (int, int);
extern int x;

int main (int argc, char* argv[]) {
    int t = func(1, 2);
    return t + x;
}

```

通过下面命令对该源代码进行编译。

```
gcc -g -o ./program ./share.so ./program.c
```

接下来，我们直接使用 GDB 调试器对主应用程序进行调试。首先将主应用程序加载到调试器中，命令如下。

```
gdb ./program -q
```

然后在应用程序的运行流程中设置断点。这里直接对共享库中的 func 函数设置断点。

```
break func
```

现在，让应用程序运行起来。

```
r
```

应用程序的运行流程在进入 func 函数之前会自动进行中断，此时我们便可以进一步查看该

函数的汇编代码信息。

```
disas /r func
```

上述命令的执行结果如图 3-26 所示。

```
(gdb) disas /r func
Dump of assembler code for function func:
0x00007ffff7bdb310 <+0>: 55      push    %rbp
0x00007ffff7bdb311 <+1>: 48 89 e5  mov    %rsp,%rbp
=> 0x00007ffff7bdb314 <+4>: 89 7d fc  mov    %edi,-0x4(%rbp)
0x00007ffff7bdb317 <+7>: 89 75 f8  mov    %esi,-0x8(%rbp)
0x00007ffff7bdb31a <+10>: 8b 45 f8  mov    -0x8(%rbp),%eax
0x00007ffff7bdb31d <+13>: 8b 55 fc  mov    -0x4(%rbp),%edx
0x00007ffff7bdb320 <+16>: 01 c2  add    %eax,%edx
0x00007ffff7bdb322 <+18>: 48 8b 05 3f 01 20 00  mov    0x20013f(%rip),%rax      # 0x7ffff7ddb468
0x00007ffff7bdb329 <+25>: 8b 00  mov    (%rax),%eax
0x00007ffff7bdb32b <+27>: 01 c2  add    %eax,%edx
0x00007ffff7bdb32d <+29>: 48 8b 05 3c 01 20 00  mov    0x20013c(%rip),%rax      # 0x7ffff7ddb470
0x00007ffff7bdb334 <+36>: 8b 00  mov    (%rax),%eax
0x00007ffff7bdb336 <+38>: 01 d0  add    %edx,%eax
0x00007ffff7bdb338 <+40>: 5d      pop     %rbp
0x00007ffff7bdb339 <+41>: c3      retq
End of assembler dump.
```

图3-26 函数func的反编译汇编代码

从代码右侧的注释中可以看到，偏移地址“0x7ffff7ddb468”和“0x7ffff7ddb470”分别对应着共享库 GOT 中的两个变量符号“x”和“y”。下面我们执行如下命令直接打印出变量“x”的实际地址。

```
p &x
```

再执行如下命令打印出主应用程序在其虚拟地址空间偏移位置“0x7ffff7ddb468”处存放的十六进制数值。

```
x 0x7ffff7ddb468
```

上面两条命令的执行结果如图 3-27 所示。

```
(gdb) p &x
$1 = (<data variable, no debug info> *) 0x60095c
(gdb) x 0x7ffff7ddb468
0x7ffff7ddb468: 0x0060095c
```

图3-27 查看变量“x”的实际偏移地址与“0x7ffff7ddb468”处存放的十六进制数值

此时你会发现，正如我们预料的那样，变量“x”的实际偏移地址与主应用程序虚拟地址空间偏移地址“0x7ffff7ddb468”处存放的十六进制数值完全相同。

GOT 在整个应用程序的动态链接过程中扮演着重要的“中转”角色，在主应用程序中使用的每一个外部共享符号都需要在 GOT 中进行注册。在 GOT 所保存的符号名与实际引用地址对应关系表中，普通的变量值符号，会直接以符号名及对应所在偏移地址的形式被存储在 GOT

中。而对于 `printf` 等函数调用过程产生的外部共享符号，GOT 则需要 PLT 结构来帮助处理这些函数共享符号对应的函数外部调用过程。

3.1.4 PLT (Procedure Lookup Table, 过程查询表)

我们之前介绍过，一个标准的 ELF 二进制文件的 GOT 结构实际上是由两部分组成的。其中一部分为用于存放值类（基本值类型、地址值等）共享符号名称与真实地址对应关系的“`.got`”表结构；另一部分为用于辅助处理函数调用类共享符号重定向过程的“`.got.plt`”表结构。这两种表结构均分别对应于 ELF 二进制文件内的同名 Section 结构。这里我们还是使用在上一节内容中编译好的主应用程序作为示例来对 PLT 结构展开介绍。首先我们还是通过 `readelf` 命令来查看该主应用程序内的符号重定向信息，输出结果如图 3-28 所示。

```
[root@iZ23ki8vt8tZ got]# readelf -r program

Relocation section '.rela.dyn' at offset 0x3f0 contains 2 entries:
   Offset             Info             Type             Sym. Value          Sym. Name + Addend
000000600928         000100000006    R_X86_64_GLOB_DAT 0000000000000000    __gmon_start__ + 0
00000060095c         000400000005    R_X86_64_COPY     000000000060095c    x + 0

Relocation section '.rela.plt' at offset 0x420 contains 2 entries:
   Offset             Info             Type             Sym. Value          Sym. Name + Addend
000000600948         000200000007    R_X86_64_JUMP_SLO 0000000000000000    __libc_start_main + 0
000000600950         000300000007    R_X86_64_JUMP_SLO 0000000000000000    func + 0
```

图3-28 主应用程序内的符号重定向信息

可以看到，位于图中下半部分的“`.rela.plt`”类型重定向信息，描述了所有被存放在“`.got.plt`”类型 GOT 结构中的符号重定向条目。存放在该表结构中的第一个符号位“`__libc_start_main`”对应的是应用程序代码中的 C/C++ 主函数；第二个符号位“`func`”则直接对应我们在主函数中进行调用的位于外部共享库中的 `func` 函数。需要注意的是，名称形式为“`.rela.*`”的 Section 结构仅用来存放对 GOT 与 PLT 中内容的描述性信息，而这两个表结构本身又分别对应其各自独立的专有 Section 结构。

事实上，之所以需要使用另外的 PLT 结构来帮助 GOT 处理与函数调用相关的符号重定向过程，是因为动态链接器每次对函数调用的符号重定向过程都需要花费较高的成本。我们将动态链接器每一次解析函数真实调用地址的过程，称为对函数调用地址的“绑定”过程。不仅如此，定义在主应用程序源代码中的所有函数调用，在程序的运行过程中并不是一定会被全部触发。比如那些专门用于异常处理的、位于不同选择分支中及属于不同业务逻辑的各类函数，在正常的业务逻辑中不会被同时触发。因此，如果在主应用程序加载时，让动态链接器一次性对所有定义在源代码中的函数调用过程都进行相应的重定向处理，那么这样会极大浪费计算机的

本地计算资源及存储资源，同时在一定程度上也降低了应用程序的运行效率。

为了解决函数调用重定向过程中存在的各种问题，PLT 便被设计出来。PLT 结构的实际作用就是对函数的调用过程进行运行时的延迟加载（类似懒加载）处理，即借助 PLT 的帮助，一个函数只有在其被真正调用到时，动态链接器才会在 GOT 中进行相应的函数调用地址重定向。

如图 3-29 所示，PLT 的内部组成结构与 GOT 类似，有所不同的是，PLT 表中的每一个独立单元都是由对应的三条汇编指令而非符号重定向地址组成的。而且这些单元的长度均保持一致，为 16 字节。整个 PLT 中的单元按照顺序被标识为 PLT0、PLT1、PLT2 等。其中 PLT0 是具有不同功能的特殊单元，其余单元则直接与 GOT 中的（位于“.got.plt”区域内）每一个函数调用重定向符号一一对应，并且这些单元中存放的三条汇编指令也是类似的。

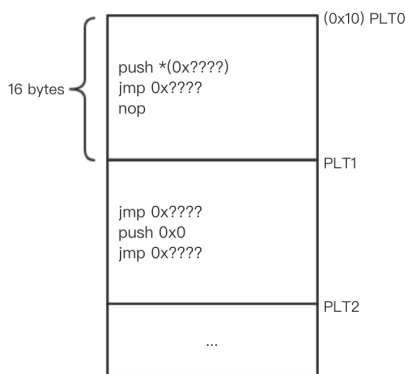


图3-29 PLT的内部组成结构

我们可以直接通过 `objdump` 命令来查看上述主应用程序内部的 PLT 结构信息。该命令的部分输出结果如图 3-30 所示。

```
Disassembly of section .plt:

000000000400460 <_libc_start_main@plt-0x10>:
400460: ff 35 d2 04 20 00    pushq 0x2004d2(%rip)
400466: ff 25 d4 04 20 00    jmpq *0x2004d4(%rip)
40046c: 0f 1f 40 00          nopl 0x0(%rax)

000000000400470 <_libc_start_main@plt>:
400470: ff 25 d2 04 20 00    jmpq *0x2004d2(%rip)
400476: 68 00 00 00 00      pushq $0x0
40047b: e9 e0 ff ff ff      jmpq 400460 <_init+0x10>

000000000400480 <func@plt>:
400480: ff 25 ca 04 20 00    jmpq *0x2004ca(%rip)
400486: 68 01 00 00 00      pushq $0x1
40048b: e9 d0 ff ff ff      jmpq 400460 <_init+0x10>
```

图3-30 主应用程序内部的PLT结构信息

可以看到,在该主应用程序所对应的 EFL 二进制文件内存在三个独立的 PLT 单元。其中被标识为“0x10”的单元是具有特殊功能的 PLT0 单元; PLT1 和 PLT2 单元分别对应于该应用程序源代码中的主函数与共享库中 func 函数的调用过程。接下来,我们将以 func 函数调用过程映射在 GOT 与 PLT 结构中的具体指令执行流程为例,来进一步介绍 PLT 的定位与功能。

首先,当应用程序开始运行时,所有位于 GOT 中的函数调用重定向符号其值都会被初始化为一个指向 PLT 结构中对应单元第二条汇编指令的地址。如图 3-31 所示,我们通过 GDB 来查看当前主应用程序内部 GOT 中函数调用重定向符号的初始值。这里直接调试主应用程序虚拟地址空间中偏移位置“0x600950”处的十六进制值,该位置为源代码中 func 函数调用过程产生的重定向符号在 GOT 中对应表项的初始值。可以很清楚地看到,该值作为偏移地址直接指向了 PLT 中 PLT2 单元的第二条汇编指令。

```
[root@iZ23ki8vt8tZ got]# gdb program -q
Reading symbols from /root/workplace/dynamic_linking/got/program...done.
(gdb) x 0x600950
0x600950 <func@got.plt>:      0x00400486
```

图3-31 通过GDB查看GOT表中函数调用重定向符号的初始值

func 函数在 Linux 系统内部的微观调用流程便是从这里开始的。首先,在该函数被初次调用时,动态链接器会通过 GOT 中符号的表项值跳转到 PLT 中对应单元的第二条汇编指令,该汇编指令在执行时会将符号在 GOT 中的位置索引值压入当前应用程序的数据堆栈中。比如对于上述主应用程序中名为“func”的函数调用符号,它在 GOT 中的位置索引值为“0x1”,即对应第二个符号位(第一个符号位为__libc_start_main 函数)。

接下来,操作系统将继续执行该单元中的第三条汇编指令,该指令会直接移动当前动态链接器的全局调用指针并指向 PLT0 单元的第一条汇编指令。之前我们提到过,PLT 结构中的 PLT0 单元具有特殊的功能,而此时在该单元中存放的三条汇编指令便发挥了真正的作用。在开始介绍这三条汇编指令前,我们首先来看一下在 GOT 中用于配合 PLT 共同完成共享库函数调用的具有动态链接功能的“.got.plt”结构的具体组成内容,如图 3-32 所示。

从图 3-32 中可以看到,在前 24 字节中,分别以每 8 字节为单位存储了 3 个用于辅助 PLT 进行函数动态链接调用的重要字段。其中存放在起始偏移位置“0x0”单元中的是 ELF 二进制文件的“.dynamic”段地址,该段被用来描述文件动态链接的相关信息;接下来在位置“0x8”对应单元中存放的是一个用于表示 ELF 二进制文件的全局唯一标识符,该标识符随后会作为参数传递给底层系统函数;最后位于偏移位置“0x10”单元中的是 Linux 下专门用于进行函数调用地址绑定的系统底层函数_dl_runtime_resolve 的函数指针,即其函数体的所在位置(该函数的

函数名根据不同的操作系统版本及类型可能会有所不同)。上面提到的全局唯一标识符将会被作为参数传递给该函数。

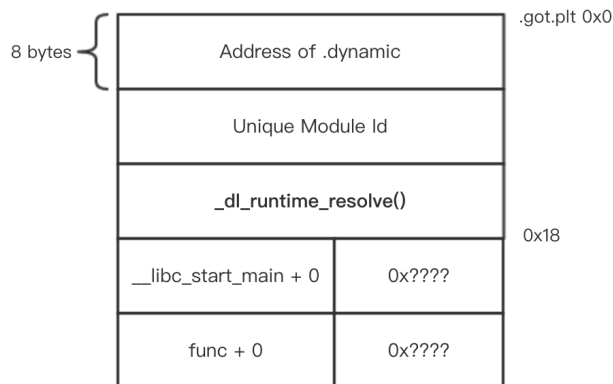


图3-32 PLT中“.got.plt”结构的具体组成内容

现在我们将目光转向 PLT0 单元中的三条汇编指令。第一条指令执行时会将存放在“.got.plt”结构中的模块全局唯一标识符压入应用程序的数据堆栈中。随后的第二条指令将会直接调用同样存放在“.got.plt”结构中的系统底层函数 `_dl_runtime_resolve` 来对相应的函数地址进行绑定操作。那么该函数是如何知道应该对 GOT 中的哪一个函数调用符号进行地址绑定呢？答案就在我们之前已经压入应用程序数据堆栈中的符号索引值，该值同样会被作为参数传递给用于进行函数地址绑定的系统底层函数，系统底层函数通过该索引值及模块的全局唯一标识符便可以完成对函数调用地址的绑定工作。当函数 `_dl_runtime_resolve` 执行完毕并解析出函数调用过程中的实际调用地址时，动态链接器会直接将 GOT 中该函数调用对应的表项值更改为从上一步获得的实际调用地址。这样，当该函数下一次被调用时，便不再需要通过 PLT 进行地址推导过程，这在一定程度上提高了代码的执行效率。PLT0 单元中的第三条汇编指令为一个 `nop` 指令，该指令没有任何的实际作用，它是为了让该单元能够满足 16 字节对齐而放置的占位指令。

上述主应用程序对应 ELF 二进制文件内 GOT 与 PLT 的完整交互流程如图 3-33 所示。基于 PLT 进行的函数调用符号重定向使得应用程序的首次加载效率和内存使用效率均有所提升。

至此，便介绍完了在 Linux 系统下 ELF 二进制文件的动态链接原理。现在我们把目光移回到主角 WebAssembly 身上。如果想要在两个 Wasm 模块之间进行类似传统应用程序与共享库之间的动态链接过程，应该如何操作？其原理是什么？我们接着往下看。

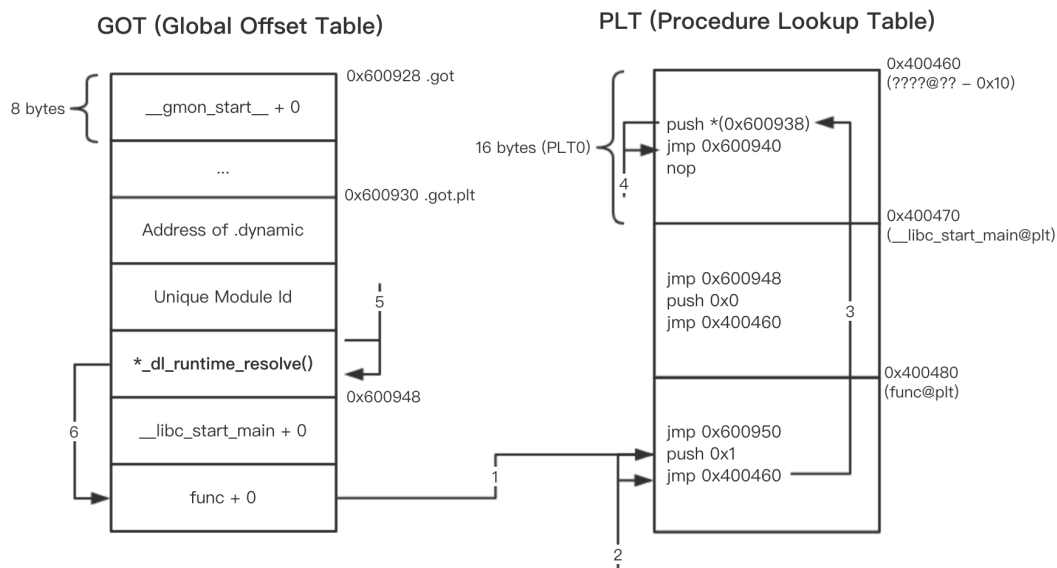


图3-33 GOT与PLT的完整交互流程（箭头表示执行流程）

3.1.5 基于表的 Wasm 模块动态链接

读到这里，你应该已经发现 Wasm 模块的内部二进制数据组成结构与标准的 ELF 二进制文件是十分相似的。两者内部均是由众多具有不同功能类型的 Section 结构组成的，甚至它们都拥有可以用来存放应用程序或模块的源代码，以及用于存放变量运行时数据值的 Section 结构。

不同于传统 ELF 二进制文件之间进行的动态链接过程，由于 Wasm 模块特殊的功能及使用环境，导致其无法基于 GOT、PLT 等类似的表结构来进行动态链接。因此，Wasm 虚拟机模拟出了一个简化版的 GOT/PLT 结构，即 Table 段结构，通过该结构我们便可以在各个模块之间使用 `call_indirect` 虚拟指令来间接地调用定义在其他模块中的函数。

接下来，我们将通过一个简单的实例来进一步了解 Wasm 模块动态链接的全过程。由于现阶段各类开发工具链都没有为模块的动态链接封装易用的 API 接口或宏参数，因此我们只能通过手动修改 WAT 代码的方式来实现 Wasm 模块的动态链接。这里直接给出两个 Wasm 模块的 WAT 代码，第一个模块的 WAT 代码如下。

```
share.wast
(module
  (memory 1) ;; 默认的共享内存段结构
  (table 2 anyfunc) ;; 默认的 Table 段结构
```

```

(export "memory" (memory 0)) ;; 导出模块的共享内存段对象
(export "table" (table 0)) ;; 导出模块的 Table 段对象
(elem (i32.const 0) $readIndex0) ;; 填充 Table 段索引位置“0”处的单元
(func $readIndex0 (result i32) ;; 被填充的函数定义
  (i32.load (i32.const 0)) ;; 获取线性内存位置“0”处的数据
)
)

```

第一个 Wasm 模块将充当传统应用程序动态链接过程中的共享库来为另一个模块提供可供调用的外部函数。

第二个模块的 WAT 代码如下。

```

program.wast
(module
  (import "env" "memory" (memory 1)) ;; 使用导入的共享内存段对象
  (import "env" "table" (table 2 anyfunc)) ;; 使用导入的共享 Table 段对象
  (export "dynamicCall" (func $f)) ;; 导出的宿主环境调用函数
  (type $void_i32 ;; 定义共享库中的函数原型
    (func (result i32))
  )
  (func $f (result i32) ;; 导出函数的定义
    (i32.store ;; 向线性内存位置“0”处写入数据“10”
      (i32.const 0)
      (i32.const 10)
    )
    (call_indirect ;; 调用 Table 段中位置“0”处的函数
      (type $void_i32) ;; 指定该函数的签名类型
      (i32.const 0) ;; 指定该函数的索引位置
    )
  )
)
)

```

该模块将充当主应用程序来主动调用定义在共享库中的 `readIndex0` 函数，该函数在调用时将从当前模块的共享线性内存段偏移位置“0”处读出一个 `i32` 类型的数据。

在这个充当主应用程序的 Wasm 模块中，我们并没有使用模块默认生成的共享线性内存段和 Table 段结构，而是通过 `import` 关键字导入了从共享库 Wasm 模块中导出的共享线性内存段对象和 Table 段对象。在这两个对象中，我们分别保存了共享库在对应段结构中已经设置好的数据值，比如共享库模块在 Table 段对象索引位置“0”处对应单元中存放的函数 `readIndex0` 的

引用指针。接下来，我们通过 `type` 关键字在该模块中声明了将要进行外部引用的共享库函数原型，就像在 C/C++ 源代码中直接通过 `extern` 关键字声明想要使用的外部符号类型一样。最后，我们定义了一个可以在外部宿主环境中进行调用的函数。该函数首先向模块当前的共享线性内存段偏移地址“0”处写入了一个 `i32` 类型的整数值“10”，然后模块通过 `call_indirect` 指令以间接的方式调用了存放在当前模块 `Table` 段结构（来自共享库模块）“0”号单元中的 `readIndex0` 函数。这里在进行函数间接调用时，需要指定该函数的签名原型以供底层虚拟机进行函数类型校验。当我们在宿主环境中调用该函数时，函数会直接返回当前模块线性内存段偏移地址“0”处存放的 32 位整数值。我们可以通过下面给出的 HTML 文件来在浏览器中运行这个简单的 Wasm 应用，在运行时还需要 HTTP 服务器来提供帮助。

index.html

```
<html>
  <head>
    <title>Dynamic Linking of WebAssembly</title>
  </head>
  <body>
    <script>
      // 定义 Buffer 转换函数
      const _toBuffer = response => response.arrayBuffer();
      // 加载 Wasm 共享库模块
      fetch('share.wasm').then(_toBuffer).then(bytes => {
        var instance = WebAssembly.instantiate(bytes, {}).then(response => {
          // 导出该模块的共享线性内存段对象和 Table 段对象
          let {memory, table} = response.instance.exports;
          // 加载主应用程序
          fetch('program.wasm').then(_toBuffer).then(bytes => {
            // 让主应用程序重用从共享库中导出的线性内存段对象和 Table 段对象
            return WebAssembly.instantiate(bytes, {
              env: {
                memory, table
              }
            })
          }).then(response => {
            console.log(response.instance.exports.dynamicCall())
            // 10
          });
        });
      });
    </script>
  </body>
</html>
```

```

</script>
</body>
</html>

```

WebAssembly 下的动态链接过程依赖 Wasm 模块所独有的几个特性：①包括共享线性内存段在内的多种段结构对象都可以被自由地导入与导出，这就使得这些段结构在上一个模块中的状态可以被下一个模块重用；②独特的 Table 段结构可以用来存放标准中允许的任何不透明值类型。另外，当我们将对应函数的 `anyfunc` 类型数据存储在 Table 段结构中时，在 C/C++ 源代码中相应的函数指针类型将会被直接映射为 Table 段内某一单元的整数索引值（从宏观上看）。这种简单的相对索引值也极大地降低了函数体在各个模块间进行转移和调用时所需要消耗的处理成本。当两个 Wasm 模块之间进行动态链接时，Wasm 虚拟机内数据栈容器与模块各个段结构间的交互流程如图 3-34 所示。

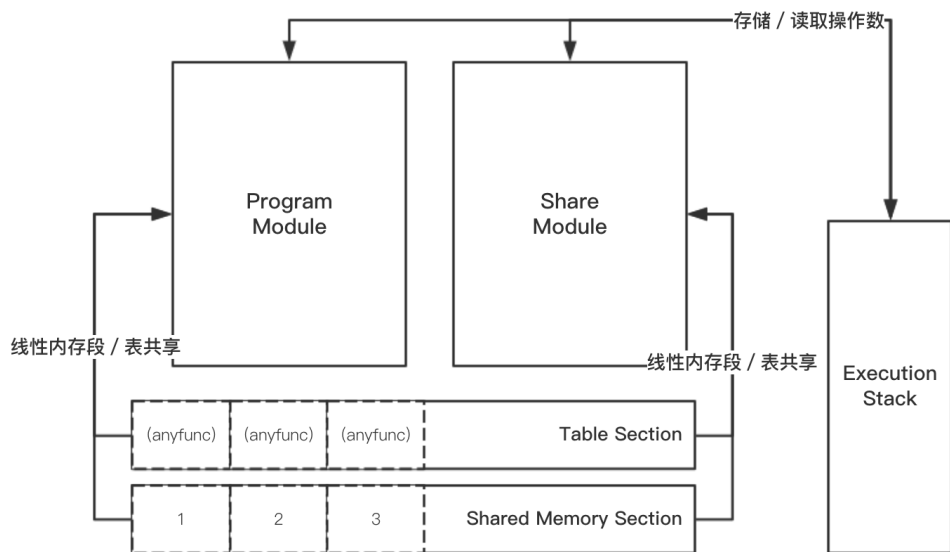


图3-34 Wasm虚拟机在处理模块动态链接时的宏观交互流程

通过使用动态链接技术，不仅能够减小各类传统应用程序或 Wasm 模块对应的二进制文件大小，而且将公用部分的代码功能提取出来也会在一定程度上减少公共资源被加载到内存中生成的副本数量，从而节省了宝贵的内存资源。（需要注意的是，Wasm 模块内的 Table 段结构虽然可用于进行模块间的动态链接过程，但实际上它很少被开发者直接通过 WAT 编码使用。相对的，编译器在编译模块时则会利用 Table 段结构来进行各种代码调用流程上的优化。）

3.2 单指令多数据流（SIMD）

通常来说，在计算机系统内部，底层的每一条汇编指令都只能对相应的一组操作数进行计算和处理。比如对于专门用于进行数学和计算的伪汇编指令 `add` 来说，当该指令每一次被执行时，CPU 都只能将对应该指令所需操作数个数的两个寄存器内的数据进行相加计算。

假如有这样一个需求：给定两个同样长度的数组结构，将这两个数组中位于相同索引位置的数字元素值进行相加求值，并将相加后得到的计算结果存储到另一个空数组中对应相同索引位置的单元内。

下面我们通过 C/C++ 等高级编程语言来实现这个需求，并通过编译器将源代码编译成依赖具体硬件架构的二进制可执行文件。此时如果将该文件进行反编译处理，并查看其内部主函数对应的汇编代码，则会发现 CPU 通过 `jmp` 指令多次在循环体中使用 `add` 指令对 `ecx` 寄存器（循环计数器）中存放的数据进行累加操作，如图 3-35 所示。

```

100000f1c: 83 7d cc 03    cmp     dword ptr [rbp - 52], 3
100000f20: 0f 8d 26 00 00 jge     38 <_main+0x7C>
100000f26: 48 63 45 cc    movsxd  rax, dword ptr [rbp - 52]
100000f2a: 8b 4c 85 ec    mov     ecx, dword ptr [rbp + 4*rax - 20]
100000f2e: 48 63 45 cc    movsxd  rax, dword ptr [rbp - 52]
100000f32: 03 4c 85 e0    add     ecx, dword ptr [rbp + 4*rax - 32]
100000f36: 48 63 45 cc    movsxd  rax, dword ptr [rbp - 52]
100000f3a: 89 4c 85 d4    mov     dword ptr [rbp + 4*rax - 44], ecx
100000f3e: 8b 45 cc    mov     eax, dword ptr [rbp - 52]
100000f41: 83 c0 01    add     eax, 1
100000f44: 89 45 cc    mov     dword ptr [rbp - 52], eax
100000f47: e9 d0 ff ff    jmp     -48 <_main+0x4C>

```

图3-35 通过`jmp`指令实现的循环操作

一般来说，这种通过汇编跳转指令模拟出的数组循环赋值操作并不会对应用程序本身的性能有任何影响。但是在某些专业级的图像处理软件或电子游戏程序中，对图片或各类 3D 素材的特效处理一般会涉及同时对大量像素点或数据进行统一的数学变换操作。这里以图片美化工具为例，当需要对图片叠加相应的滤镜效果时，一般会通过应用程序对图片中每一个像素点使用对应特效的卷积核进行卷积处理来实现。此时如果仍然通过循环的方式依次对图片中的每一个像素点进行相同的卷积处理，则这样不免显得有些“鸡肋”且低效。那么是否有一种方法可以让 CPU 同时对数据向量中的每一个数据都进行同样的运算操作呢？答案就是我们即将要介绍的 SIMD 技术。

SIMD（Single Instruction, Multiple Data，单指令多数据流）描述了一种可以实施并行计算的计算机体系架构。在该体系中，计算机内部拥有多个数据处理单元，这些单元可以在同一时

间内对一簇数据集中的每一个数据都进行相同的处理操作。从某种角度来看，这种在同一个时间片内对多路数据进行同步计算的过程可以算作真正意义上的并行计算。下面我们对传统计算机内部的数据处理方式与基于 SIMD 架构的计算机其内部的数据处理方式进行对比。

如图 3-36 所示的是传统计算机内部的数据处理方式。可以看到，在当前的线性内存段中存放着 4 个位于连续地址上的整数“1”“3”“6”“4”。此时计算机将要进行的操作是：将这 4 个整数与数字“3”分别进行数学乘积计算，并将最后的计算结果重新写入线性内存中。这个看似十分简单的操作，实际上却需要 CPU 至少执行 12 条指令才可以完成。从整体上看，CPU 需要通过 4 轮循环来分别处理这 4 个整数，其中每一轮循环所需要执行的指令有：首先，CPU 通过用于从内存中读取数据的 load 指令将操作数读入相应的寄存器（R1 寄存器）中；然后，CPU 通过乘法指令 mul 对位于两个寄存器中的数据进行处理，计算结果被存放到另一个新的寄存器（R3 寄存器）中；最后，CPU 通过相应的内存写指令将计算结果重新写回内存。

如图 3-37 所示的是基于 SIMD 架构的计算机其内部的数据处理方式。对于同样的一段代码逻辑，在支持 SIMD 架构的处理器下只需要执行 3 条指令即可完成，每一条指令都可以在同一时间内对多个操作数并行地进行计算和处理。首先，通过 SIMD 指令集内专门用于从内存中读取数据的 simd_load 指令，CPU 可以一次性（在一个 CPU 时钟周期内）将位于连续内存地址上的 4 个整数读取到相应的 SIMD 专用向量寄存器中。然后，通过专门用于 SIMD 向量乘法计算的 simd_mul 指令，一次性对寄存器内的 4 个操作数进行乘法计算，计算结果会被保存到另一个向量寄存器中。最后，通过专门用于向线性内存中写入向量数据的 SIMD 指令 simd_store，直接将 4 个计算结果值一次性写入线性内存中的一块连续地址上。（上述指令均为伪指令）

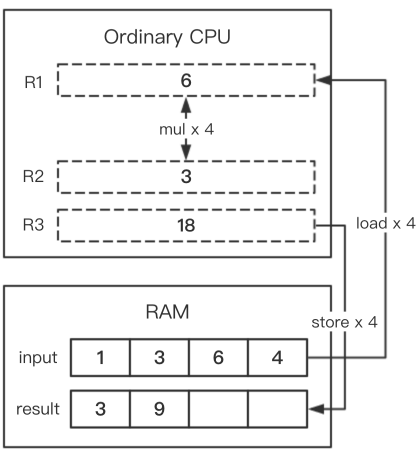


图3-36 传统计算机内部的数据处理方式

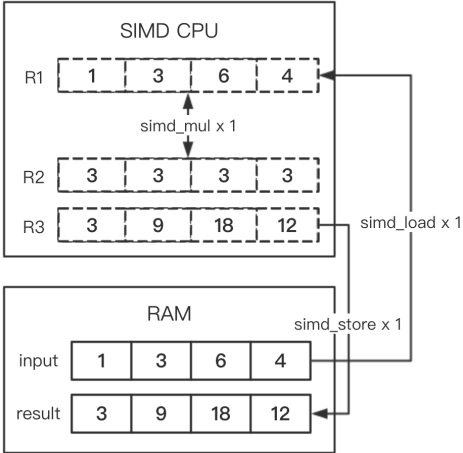


图3-37 基于SIMD架构的计算机内部的数据处理方式

3.2.1 SIMD 应用

本节我们将通过构建一个简单的 C/C++ 应用程序来了解传统 SIMD 技术的使用方式。这里我们基于 SSE 指令集来构建该 SIMD 应用。SSE (Streaming SIMD Extension, 流式 SIMD 扩展) 指令集是由 Intel 公司于 1999 年在其奔腾 3 系列的 CPU 处理器中设计并随后推出的。在 SSE 指令集中包含了一系列专门用于 SIMD 技术的处理器指令, 这些指令中的绝大部分可以直接被使用在由单精度浮点数组成的数据向量上。SSE 指令集从诞生之初到现在, 已经发布了多个版本, 每一个新发布的版本相比前一个版本都增加了更多的数据运算指令。2006 年发布的 SSE 4 版本是现阶段 SSE 指令集的最新可用版本, 整个 SSE 指令集的指令数量已经超过 200 个。

MMX 是 Intel 公司在推出 SSE 之前设计的一个专门用在其 x86 架构 CPU 上, 且支持 SIMD 技术的指令集。与 SSE 指令集相比, MMX 指令集存在多种数据处理及寄存器交互的问题, 进而导致其逐渐被市场所淘汰。首先, MMX 直接重用了 CPU 内部正常的线性浮点数寄存器, 但由于该寄存器不能同时存放普通浮点数, 以及 SIMD 下的浮点数向量, 因此导致操作系统无法高效地运行混有 SIMD 向量处理逻辑的应用程序。其次, 基于 x86 架构, 在 MMX 对应的处理器中只单独设置了 8 个专门用于存放整型向量数据的 64 位寄存器, 这对于某些包含有较多大数值元素的数据向量来说, 其数据处理过程可能较为烦琐。

我们回过头再来看 SSE 所对应的底层硬件技术体系。SSE 在其最初的架构设计中新增了 8 个 128 位长度的寄存器, 其名称分别为 XMM0, ..., XMM7。这 8 个寄存器可以在不同 SSE 版本指令集中相应指令的作用下存放各种不同的数据类型。比如对于 SSE 1 版本的指令集来说, 这 8 个寄存器中的每一个寄存器都只能存放由 4 个 32 位单精度浮点数组成的数据向量; 而对于 SSE 2 版本的指令集来说, 可以存放多种数据类型。不仅如此, 在 64 位处理器下, SSE 还会自动启用另外的 8 个寄存器 (XMM8, ..., XMM15) 以支持更长的向量处理长度。事实上, 为了不影响普通指令的执行流程, 在正常的情况下操作系统并不会主动启用这 16 个寄存器。因此, 在实际的应用开发过程中, 我们需要在源代码中声明特定的、编译器能够识别的向量数据结构, 并在编译时为编译器指定 SSE 指令集版本, 这样操作系统才能明确地启用 SIMD 特性。

关于 SIMD 技术的背景就介绍到这里, 接下来我们将在实际的应用中进一步体会这些 SIMD 专用的寄存器及对应的部分 SSE 指令。下面给出这个简单 SIMD 应用的 C++ 源代码。

```
simd.cc
#include <iostream>

using namespace std;
```

```
// 定义向量类型，v4sf 为一个包含有 4 个单精度浮点数的数据向量
typedef float v4sf __attribute__((vector_size (16)));
// 定义 Matrix 类
class Matrix {
public:
    // 利用友元函数重载运算符 "<<"
    friend ostream& operator<<(ostream& os, const Matrix& dt);
    void setData (v4sf v) {
        this->data = v;
    }
    v4sf getData () const {
        return this->data;
    }
private:
    v4sf data;
};

ostream& operator<<(ostream& os, const Matrix& dt) {
    v4sf _t = dt.getData();
    os << _t[0] << " "
        << _t[1] << " "
        << _t[2] << " "
        << _t[3];
    return os;
}

int main (int argc, char* argv[]) {
    // 声明并初始化一些 v4sf 类型的向量数据
    v4sf a = {1.0, 2.0, 3.0, 4.0};
    v4sf b = {2.0, 3.0, 4.0, 5.0};
    v4sf c = a + b;
    // 调用编译器内置的向量数据处理方法，这些方法会直接调用 CPU 的底层 SSE / MMX 指令
    v4sf d = __builtin_ia32_sqrtps(c);

    Matrix m1, m2;
    m1.setData(c);
    m2.setData(d);
    // 打印 Matrix 类中的向量数据
    cout << endl
        << m1 << endl
```

```

    << m2 << endl
    << endl;
}

```

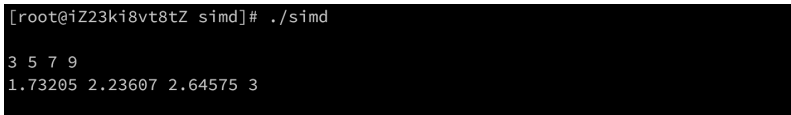
在上面的代码中，我们首先通过 `typedef` 关键字定义了一个名为“v4sf”的用于存储向量数据的特殊类型。因为每一种编译器所规定的向量类型数据声明方式都有所不同，这里使用 GCC 编译器下的 `__attribute__` 描述符来声明该向量类型的具体信息。其中 `float` 关键字指定了向量类型中存储元素的基本类型，即单精度浮点数。`vector_size` 属性描述了该向量类型的总长度，即 16 字节。经过换算得知，在该向量类型对应的变量中，我们可以一次性存入 4 个连续的单精度浮点数。

接下来，我们定义了一个向量处理类 `Matrix`，在该类结构中重载了输出运算符“<<”，使其可以直接作用于之前的向量类型数据。当应用程序开始运行时（对应主函数），首先创建并初始化了两个 `v4sf` 类型的向量数据，并让它们进行数学相加运算。计算结果被直接传递给了一个名为“`__builtin_ia32_sqrtps`”的编译器内置函数，该函数会对向量中的每一个元素进行平方根求值。最后我们将函数的计算结果赋值给了新创建的 `Matrix` 类对象，并通过重载后的输出运算符将该结果打印出来。

当源代码编写完成后，使用如下命令对该源代码进行编译。注意：这里使用的是“g++”命令。还需要在命令中添加“-msse”参数，以告知编译器我们需要使用的 SIMD 指令集类型及版本。

```
g++ -msse -o simd ./simd.cc
```

编译完成后，我们可以直接在本地运行该应用程序，运行结果如图 3-38 所示。图中第一行数据为向量“a”与向量“b”在进行数学相加运算后得到的结果。在计算过程中，位于两个向量中相同索引位置处的元素值会直接相加，而计算后得到的结果值将会被直接存放到向量“c”中同样索引位置处的单元中。第二行数据为向量“c”中的元素在经过 `__builtin_ia32_sqrtps` 函数处理后得到的平方根计算结果。



```

[root@iZ23ki8vt8tZ simd]# ./simd
3 5 7 9
1.73205 2.23607 2.64575 3

```

图3-38 上述SIMD应用程序的运行结果

为了进一步验证我们编译好的应用程序是否真正使用了 SIMD 指令，这里通过 `objdump` 命令来查看其反编译汇编代码。如图 3-39 所示，在应用程序主函数开始部分的汇编代码中，我们可以发现很多名为“`movaps`”的 SSE 指令。该指令名是由其功能描述语句“`MOVE four Aligned`

Packed Single precision”中的几个大写字母组合而成的，其作用是将 SIMD 寄存器中已经打包好的 4 个单精度浮点数移动到其他寄存器或线性内存中。对于这里提到的“打包”，可以将其理解为：从整体上将多个数据视作一个向量，并且对该向量进行的任何操作都会直接作用于该向量内部的所有元素。

```
000000000400a19 <main>:
400a19:    55                push    rbp
400a1a:    48 89 e5          mov     rbp,rsi
400a1d:    48 83 ec 40       sub     rsp,0x40
400a21:    0f 28 05 b8 01 00 00 movaps  xmm0,XMMWORD PTR [rip+0x1b8]
400a28:    0f 29 45 f0       movaps  XMMWORD PTR [rbp-0x10],xmm0
400a2c:    0f 28 05 bd 01 00 00 movaps  xmm0,XMMWORD PTR [rip+0x1bd]
400a33:    0f 29 45 e0       movaps  XMMWORD PTR [rbp-0x20],xmm0
400a37:    0f 28 45 e0       movaps  xmm0,XMMWORD PTR [rbp-0x20]
400a3b:    0f 28 4d f0       movaps  xmm1,XMMWORD PTR [rbp-0x10]
400a3f:    0f 58 c1          addps   xmm0,xmm1
400a42:    0f 51 c0          sqrtps  xmm0,xmm0
```

图3-39 应用程序主函数部分对应的汇编代码

最后，我们通过 GDB 来查看当应用程序处于运行状态时，计算机 CPU 内部各个 SIMD 寄存器的数据状态。如图 3-40 所示，列出了从 XMM0 到 XMM15 共 16 个 SSE 寄存器中存放的数据内容。由于寄存器本身并不知道存放在其内部的向量数据类型及元素个数，因此 GDB 会帮助我们自动按照元素的数据类型将向量元素的可能取值分为多个不同的数据单元。比如图中的“v4_float”单元表示，如果该寄存器内部存放的是 4 个单精度浮点数，则对应向量中各元素的取值情况将会如何。其他单元如 v2_double、v16_int8 等均表示向量的可能组成方式，它们的元素组成形式与 v4_float 类似，可以依此类推。在图 3-40 中列出的所有寄存器中，位于列表最后的 mxcsr 寄存器与其他寄存器不同，该寄存器的长度仅为 32 位，其主要作用是保存在应用程序中使用到的 SSE 指令涉及的各种控制及状态性标志位。

```
xmm0    {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 11 times>, 0xff, 0x0, 0x0, 0
xmm1    {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 11 times>, 0x1, 0x0, 0x0, 0x
xmm2    {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x8000000000000000, 0x8000000000000000}, v16_int8 = {0x4e, 0x53,
0x61636f6c, 0x6635656c, 0x74656361}, v2_int64 = {0x61636f6c3674534e, 0x746563616635656c}, uint128 = 0x746563616635656c6163
xmm3    {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0,
xmm4    {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0,
xmm5    {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0,
xmm6    {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0,
xmm7    {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0,
xmm8    {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0,
xmm9    {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0,
xmm10   {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0,
xmm11   {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0,
xmm12   {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0,
xmm13   {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0,
xmm14   {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0,
xmm15   {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0,
mxcsr   0x1f80 [ IM DM ZM OM UM PM ]
```

图3-40 XMM寄存器中存放的数据内容

从宏观角度来看，针对 SIMD 架构具有的一次性（一条指令）可以处理多个数据值的特性，我们认为它是具有数据并行处理能力的。作为同样可以用于提升应用程序运行效率的一项技术，基于高并发实现的多线程技术与 SIMD 技术在不同的底层硬件情况下，其各自的数据处理效率有着怎样的差别，以及这里提到的并行与并发又有着怎样的关联和不同，我们继续往下看。

3.2.2 并行与并发

从宏观上看，不论是并行还是并发，它们都是一种数据处理单元（DPU）可以同时处理多个数据源（DS）中的数据的过程。但是从其各自本身的数据处理方式来看，它们却有着巨大的差别。如图 3-41 所示，我们先来看“并行”这个概念是如何体现在数据处理单元与数据源之间的交互模式上的。



图3-41 并行的数据处理方式

可以看到，并行的数据处理方式是多个数据源分别对应于不同的数据处理单元，多个数据处理单元同时对各数据源中的数据进行处理，并且各个数据处理单元之间不发生交互和耦合。

我们再来看一下并发的数据处理方式，如图 3-42 所示。可以看到，多个数据源同时传递数据给唯一的数据处理单元进行处理，而数据处理单元则快速地以“轮流交错”的方式来处理位于不同数据源中的数据。由于现代 CPU 具有的高效运算能力，这种数据处理方式会给人一种多个数据源中的数据被数据处理单元同时进行处理的感觉。



图3-42 并发的数据处理方式

实际上，传统的多线程技术既可以在并行模式下实现，也可以在并发模式下实现。而究竟

选择哪种模式，则要取决于 CPU 在物理层面所具有的独立核心数量。在具有多个核心的 CPU 上，每一个独立的核心都可以单独控制一个线程来进行任务处理，这样便真正地实现了并行模式下的多线程技术；而在单核 CPU 上便只能以并发模式来使用多线程技术。但这并不意味着在所有的情况下并行模式都要好于并发模式，具体使用哪种模式则需要根据特定的业务场景来进行选择。相对而言，SIMD 技术只能以并行的数据处理方式来实现。即在单一 SIMD 指令执行的情况下，对于一个向量中的所有数据元素，其值必定是被同时修改的。而不同的是具体 CPU 下的 SIMD 指令集，以及相关寄存器的实现方式。

3.2.3 费林分类法

Michael J.Flynn（音译：费林）于 1996 年根据不同计算机中可用的并发指令流及对应数据流的数量，将计算机的底层体系架构划分为 4 种类型，这种分类方法被后人称为“费林分类法（Flynn's taxonomy）”。按照费林分类法中的描述，传统计算机的底层体系架构可以被分为如下 4 种类型。

SISD（单指令单数据流）

如图 3-43 所示，在基于 SISD 架构实现的计算机中，每一个 CPU 上的控制单元一次只能从线性内存中读取出一个指令流。接下来控制单元会生成适当的控制信号来指导 CPU 上的单个处理单元（PU）对单个数据流中的数据进行处理。一般来说，部分老式的只有单核心处理器的个人计算机会采用这种体系架构。

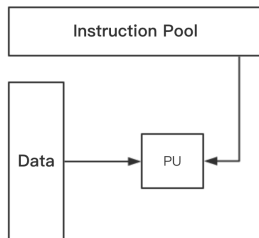


图3-43 SISD架构的数据交互模式

SIMD（单指令多数据流）

顾名思义，SIMD 架构即每一个 CPU 上的控制单元一次只能从线性内存中读取出单个指令流，但接下来控制单元会生成适当的控制信号来指导 CPU 上的多个处理单元（PU）对多个数据流中的数据进行并行处理。需要注意的是，多个处理单元并不等于多个 CPU 处理核心。在多核心的 CPU 上，多个核心的功能是完全相同的，它们都可以执行任何一个 SSE 或 MMX 指令

集中的 SIMD 指令。一般来说，SIMD 架构经常会被应用在诸如 GPU 图像处理、向量计算等需要大量并行计算的场景中。SIMD 架构的数据交互模式如图 3-44 所示。

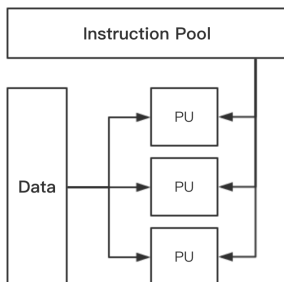


图3-44 SIMD架构的数据交互模式

MISD（多指令单数据流）

MISD 即 CPU 上的控制单元一次性可以从线性内存中读取出多个指令流，接下来控制单元会生成适当的控制信号来指导 CPU 上多个核心的多个处理单元（PU）对单个数据流中的数据进行处理。在 MISD 架构下，多个指令对应的各个处理过程会并行进行。从某种程度上看，这种体系架构也属于一种数据并行处理方式，即单一数据对应多个处理进程。基于其“多处理机”的特性，MISD 架构通常被应用在航天飞机的飞行控制系统中，以实现故障的容错机制。MISD 架构的数据交互模式如图 3-45 所示。

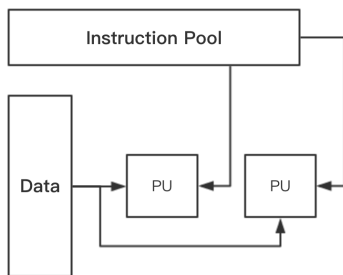


图3-45 MISD架构的数据交互模式

MIMD（多指令多数据流）

MIMD 结合了 SIMD 与 MISD 两种架构的特性，即 CPU 上的控制单元一次性从线性内存中读取出多个指令流，接下来控制单元会生成适当的控制信号来指导 CPU 上多个核心的多个处理单元（PU）同时对多个数据流中的数据进行处理。MIMD 架构的应用场景十分广泛，在包括“超级计算机”在内的对并行计算有着较高要求的各种场景中都可以使用 MIMD 架构。MIMD 架构的数据交互模式如图 3-46 所示。

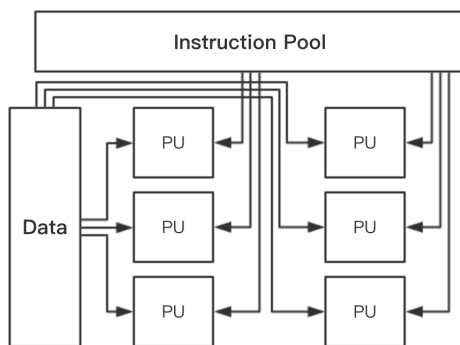


图3-46 MIMD架构的数据交互模式

3.2.4 SIMD.js & TC39

SIMD.js 是一套用于在浏览器中使用 CPU 底层 SIMD 技术的 JavaScript 接口，是由 Intel、Google 和 Mozilla 三家共同提出并开发的。在这套 API 中引入了可以直接用于进行 SIMD 计算的向量数据类型，以及相关计算和数据处理函数。SIMD.js 在其底层实现上，同时支持 x86 架构下的 SSE 指令集和 ARM 架构下的 NEON 指令集，以保证使用 SIMD.js 构建的 Web 应用程序具有较高的可移植性。不仅如此，在 Emscripten 工具链的帮助下，我们还可以将使用了 SIMD 指令的 C/C++ 应用程序直接编译成能够在浏览器上运行的 ASM.js 应用。此时应用程序中的原始 SSE/NEON 指令将会被 Web 端的上层 SIMD.js 函数代替。

SIMD.js 的诞生源于 2013 年 John McCutchan 通过 Dart 语言实现的一个可以运行在 Web 平台上的高性能 SIMD 指令集接口。在该接口中包含两种不可变的 SIMD 类型：Float32x4（包含 4 个 32 位的单精度浮点数）和 Int32x4（包含 4 个 32 位的整数），这两种类型的实例将会在 Dart 代码的优化阶段被直接映射到对应的各个 SIMD 寄存器上。该指令集接口的实现方式随后被 TC39 委员会采纳，并被作为 SIMD.js 实现在 Google 和 Mozilla 的 V8 及 SpiderMonkey 引擎中。

但事情并不是一帆风顺的。由于直接在 JavaScript 引擎中实现需要具有较高可用性，并且在各类型指令集上均保持效果一致的 SIMD 指令其难度较高，因此 TC39 最终决定将已经处于 Stage 3 评估阶段的 SIMD.js 从计划提案中移除，同时将 Web 平台上的可用 SIMD 指令实现在 WebAssembly 标准中。由于当前大部分浏览器不再支持 SIMD.js 接口，因此我们只能在 Firefox 的实验专用浏览器 Nightly 中来体验该技术的具体应用方法及性能优化效果。下面给出的是一段基于 SIMD.js 接口编写的 JavaScript 源代码，我们将比较基于 SIMD.js 与普通循环语句实现的同一段数据处理逻辑在性能上的差异。

```
// 定义载有数据的 TypedArray 类型容器
let x = new Int8Array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]);
let y = new Int8Array([16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]);
// 定义用于存放结果数据的 TypedArray 类型容器
let z = new Int8Array(16);
const _start = performance.now();
// 构建一个大循环结构
for (let i = 0; i < 1e+6; i++) {
  // 从 TypedArray 容器中生成向量数据
  let _x = SIMD.Int8x16.load(x, 0);
  let _y = SIMD.Int8x16.load(y, 0);
  // 将两个向量中的数据进行求和运算
  let _z = SIMD.Int8x16.add(_x, _y);
  // 将运算结果存入原始的 TypedArray 容器中
  SIMD.Int8x16.store(z, 0, _z);
}
console.log("Time Used: " , (performance.now() - _start).toFixed(2), "ms");
```

从上面的源代码中可以看到，我们首先定义了两个用于存放原始数据和一个用于存放运算结果的 `TypedArray` 类型容器。接下来在一个大循环结构中，我们将容器中的数据转换成 `SIMD.js` 中特定类型的向量数据，并对两个向量中的数据进行求和运算，然后将运算结果通过特定的方法存放到用于保存计算结果的 `TypedArray` 类型容器中。在整个源代码中，我们将使用 `SIMD.js` 相关接口进行数据处理的部分循环执行了 10 万次，并记录下浏览器花费的时间。在 `Firefox Nightly` 浏览器中，该 `SIMD.js` 应用的运行结果如图 3-47 所示。



图3-47 上述SIMD.js应用的运行结果

现在，我们通过 `TypedArray` 自带的 `map` 方法来实现同样的数据处理逻辑，同时记录下整个循环结构执行所花费的时间。对应的 `JavaScript` 代码如下。

```
let x = new Int8Array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]);
let y = new Int8Array([16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]);
let z = new Int8Array(16);
const _start = performance.now();
for (let i = 0; i < 1e+6; i++) {
  // 通过 map 函数依次对两个 TypedArray 容器中的数据进行求和运算，并将最后的运算结果返回到结果容器中
  z = x.map((item, index) => {
    return item + y[index];
  });
}
```

```
});
}
console.log("Time Used: ", (performance.now() - _start).toFixed(2), "ms");
```

上面这段 JavaScript 源代码在浏览器中的运行结果如图 3-48 所示。



图3-48 基于传统循环结构实现的应用运行结果

可以看到，从整个循环结构的执行时间来看，基于 SIMD.js 实现的向量数据处理逻辑其运行效率要高于传统的数据处理方式。但是由于与实际业务相比测试用例的针对性较强，因此并不能一概而论。

3.2.5 WebAssembly 上的 SIMD 扩展

为了能够取代 SIMD.js 而继续在 Web 端应用中支持 SIMD 技术，WebAssembly 在其 Post-MVP 计划中加入了关于整合 SIMD 相关类型与虚拟指令的提案。在这份提案中，官方小组（WWG & WCG）为 Wasm 虚拟机新加入了一个名为“v128”的值类型（对应的二进制编码为“0x7b”），该类型以直接映射到 SIMD 所对应的 128 位寄存器中各个比特位的方式，来表示一个可以用于 SIMD 指令的向量值类型。与传统 SIMD 指令集的执行方式类似，对于每一个 128 位寄存器内的数据，我们都需要了解其组成方式，包括单个元素数据的具体类型，以及向量中该元素的总个数。新增的 v128 类型便可以帮助做到这一点，它可以通过多种方式来将数据打包成不同的组成形式。在 Wasm 虚拟机中可用的 SIMD 数据组成形式如下。

v8x16

v8x16 表示将一个 SIMD 寄存器中的 128 位数据分为 16 个通道，每个通道存放一个 8 位长的数值。该组成形式可用于存放整型数据，对应的 Wasm 虚拟机类型构造符为“i8x16”。

v16x8

v16x8 表示将一个 SIMD 寄存器中的 128 位数据分为 8 个通道，每个通道存放一个 16 位长的数值。该组成形式可用于存放整型数据，对应的 Wasm 虚拟机类型构造符为“i16x8”。

v32x4

v32x4 表示将一个 SIMD 寄存器中的 128 位数据分为 4 个通道，每个通道存放一个 32 位长的数值。该组成形式可用于存放整型和浮点型数据，对应的 Wasm 虚拟机类型构造符为“i32x4”和“f32x4”。

v64x2

v64x2 表示将一个 SIMD 寄存器中的 128 位数据分为 2 个通道，每个通道存放一个 64 位长的数值。该组成形式可用于存放整型和浮点型数据，对应的 Wasm 虚拟机类型构造符为“i64x2”和“f64x2”。

可以看到，这里用于定义 SIMD 向量数据的类型构造符，在原有普通类型构造符的基础上又新增了一个参数，用于指定向量数据类型中的并行通道个数。并且该参数与对应数据类型长度的乘积，直接等于 Wasm 虚拟机中底层 SIMD 寄存器的最大宽度，通常为 128 位。另外，在 Wasm 中一系列基于 SIMD 实现的数据处理指令（如 i16x8.add、v8x16.shuffle 等虚拟指令）其组成方式与传统的 Wasm 虚拟指令相同，只是该类指令的调用参数及返回值必须均为 v128 类型。而有所不同的是，每一个用于支持 SIMD 技术的 Wasm 虚拟指令其所对应的二进制 OpCode 代码都需要另外再增加一个十六进制值“0xfd”来作为其组成前缀。

提示：截至本书完稿之日，该提案还未被任何浏览器实现，因此这里无法给出相应的实践过程。且本节介绍的 SIMD 标准可能会在后续的浏览器实现过程中发生变化。

第4章

深入 LLVM 与 WAT

在前两章的内容中，我们主要围绕 WebAssembly 的核心设计原理，以及部分与其相关的功能特性，从概念和实践两方面进行了介绍。在本章中，我们将首先抛开 Wasm 本身，来聊聊与其紧密相关的另一个话题——LLVM。

4.1 LLVM——底层虚拟机

LLVM 是英文全称“Low Level Virtual Machine”的缩写形式，其中文名称为“低层次虚拟机”。最初，LLVM 是由 Chris Lattner 和 Vikram Adve 两人于 2000 年 12 月在伊利诺伊大学着手研发的一套综合性的软件工具链。在这套工具链中，包含了众多可用于开发语言编译器、链接器、调试器等操作系统底层基础构建工具的相关组件。比如著名的 Clang 编译器便是基于 LLVM 构建的，相比于 GCC（the GUN Compiler Collection，GUN 编译器集合）所包含的一系列语言编译器而言，其独有的如可插入型优化器、支持源代码链接时优化等特性均是基于 LLVM 工具链来提供的。除 LLVM 所提供的一系列可重用组件之外，还有一个最重要的特性便是其内部独特的编译架构，基于 LLVM-IR 实现的全新编译链路也为我们如何更好地开发编译器等基础工具提出了新的思考。现在 LLVM 已经从工具集的名称转变为一个独立的品牌名，该品牌专指 LLVM 工具链，以及基于该工具链所构建的一系列子项目，如各类语言编译器、调试器、功能组件等。整个 LLVM 品牌和对应的软件项目均由专门的 LLVM 基金会负责管理。

LLVM 在开发初期被设计定位为是一套具有良好接口定义的可重用组件库，其中组件所具有的通用性功能则方便它们在各个软件中使用。当时，开源编程语言的编译器通常被设计成单一实现的模式，即从编译器前端的词法分析器到编译器后端的机器码生成器，全部被从头到尾实现在单一的二进制可执行文件中。因此，如果想在其他应用程序中使用 GCC 等静态编译器中的

解析器，来单独实现对代码的静态分析，则需要将这部分功能从编译器的链路中抽离、重构后再嵌入应用程序中使用，无疑成本将是巨大的。不仅如此，虽然 Chrome V8 等脚本语言的执行引擎可以被嵌入其他独立的应用程序中使用，但是我们仍然无法在应用程序代码中直接重用这些编译引擎的内部代码。

从另一方面来看，除编译器本身单一的实现方式以外，围绕流行编程语言的编译器实现也逐渐变得两极化。语言编译器的实现通常分为两类：一类是 GCC 等静态编译器；另一类则是以提供代码解释执行为主的 JIT 编译器。事实上，我们很难看到同时支持两类语言编译方式的编译器实现。这是因为实现这两类编译器的技术原理各不相同，由于使用了单一实现的方式来构建编译器，因此导致它们之间很少有能够被共享的代码，而重新独立开发两套编译器的成本又是巨大的。

但是在过去的十年时间里，LLVM 已经帮助我们极大地改变了这种状况。LLVM 最初是为 C 和 C++ 语言设计的专门用于研究静态和动态编程语言编译技术的基础设施工具，而现在它已经被作为通用的编译器工具链来实现各类静态或动态编译语言（如 ActionScript、C#、Common Lisp 等语言）的编译器前端，以及各类硬件平台上的 LLVM 编译器后端。不仅如此，LLVM 还在一定程度上取代了具有特殊用途的 DSL 编译器，比如在苹果 OpenGL 堆栈中专用的运行时引擎，以及 Adobe AE 软件中的图像处理库也有 LLVM 的身影。

4.1.1 传统的编译器架构

对于传统的静态语言编译器（即需要进行 AOT 编译），我们通常会使用最流行的“三段式”链路结构来进行开发。如图 4-1 所示，三段式结构即对应编译器的三个最重要组成部分，它们分别为：用于分析源代码的编译器前端、用于优化中间代码（IR）的优化器和用于生成平台相关机器码的编译器后端。其中，编译器前端又由用于分析源代码中各组成元素具体类型的词法分析器（Lexer）子程序和用于分析源代码各部分语法结构的语法分析器（Parser）子程序组成。编译器前端对源代码进行一系列的分析和处理后，会向位于链路中间的优化器输入用于表示源代码基本语法逻辑的 AST 数据。优化器会对输入进来的 AST 数据进行相应的优化处理，以便最大程度地提升源代码的执行效率。经过优化后的中间表示（IR）代码随后会被送往位于链路末端的编译器后端。在这里，编译器会进行指令选择、寄存器分配等操作，最后将 IR 转换为平台相关机器码，并选择性地通过链接操作生成直接可用的二进制可执行文件或共享库文件。

当一个编译器需要支持多种类型的源语言或目标平台体系架构时，其三段式链路结构设计优势便体现出来。首先，我们要保证位于链路结构中间的优化器只能接收符合特定格式的 AST

数据作为其输入，同时也只能输出特定格式的 IR 数据。这样当需要为编译器新增一种其他类型的源语言时，只需重新设计和编写编译器前端即可。只要确保用于该语言的编译器前端能够生成符合特定格式的 AST 数据，并将其送往优化器中进行处理，整个编译器就可以正常运行。同理，如果想为编译器新增一个具有不同指令集类型的目标平台，则只需重新编写编译器后端即可。从整体上讲，这种分段式的链路结构使得我们可以充分地重用编译器的各个组成部分。但遗憾的是，在 LLVM 出现之前，各编程语言编译器中优化器的实现并没有采用统一的 AST 和 IR 数据结构，这导致对编译器链路各组成部分的重用仍然十分困难。

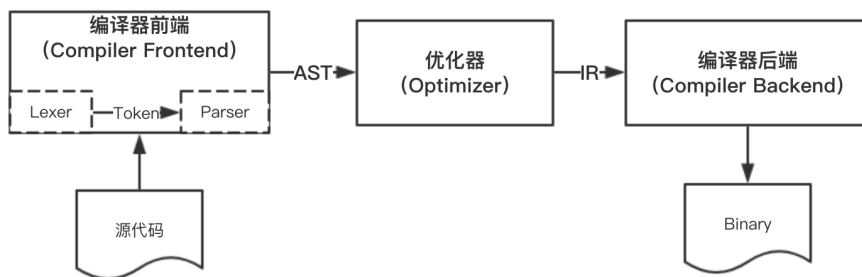


图4-1 传统的三段式编译器链路结构

另外，编译器的分段式结构设计也使其开发分工更加明确。比如擅长编译器前端设计的开发人员，可以更加专注地进行编译器前端的具体实现，而不用考虑应该为后续的优化器和编译器后端预留何种类型的资源，以及进行何种配置。这对于将要投入商业运作中的编译器软件来说无疑是最有利的。

除此之外，编译器的分段式结构设计也为编译器的相关开源社区注入了一股新的力量，有更多的编译器开发爱好者逐渐加入进来，致力于改进现有的开源编译器，他们提交性能更高的代码实现、修复漏洞，甚至提供新的前、后端实现。这无疑使开源编译器变得更加稳定，目标代码的性能也随着编译器的逐渐优化而不断提高。

4.1.2 LLVM 中间表示层

在整个 LLVM 工具链中，最重要的设计思想便是其用于表示编译器中间状态的代码格式——LLVM-IR。在一个基于 LLVM 实现的编译器链路中，位于链路中间的优化器将会以 LLVM-IR 作为其统一的输入与输出数据格式。在整个 LLVM 项目中，扮演着重要角色的 LLVM-IR 被定义成一类具有明确语义的轻量级、低层次的类汇编语言，其具有足够强的表现力和较好的可扩展性。通过更加贴近底层硬件的语义表达方式，它可以将高级语言的语法清晰地映射到其自身。

不仅如此，通过其语义中提供的明确的变量类型信息，优化器还可以对 LLVM-IR 进行进一步的深度优化。

与 Wasm 模块一样，整个 LLVM-IR 代码具有三种表示形式：存放于内存中可直接用于编译器的二进制中间代码、存放于磁盘等外部存储介质中的二进制比特码，以及与 WAT 格式类似的可读文本代码。这三种表示形式的 LLVM-IR 代码是完全等价的，我们可以通过 LLVM 工具链提供的相关命令行工具将它们在这几种状态之间进行转换。如下给出了一段可读的 LLVM 中间表示代码（摘自 LLVM 官方文档）。

```
; 定义一个全局字符串常量，同时指定该常量中各个字符数据的类型，并指定相应的属性修饰符
@.str = private unnamed_addr constant [13 x i8] c"hello world\0A\00"

; 声明外部函数 puts
declare i32 @puts(i8* nocapture) nounwind

; 对主函数 main 的定义过程
define i32 @main() {    ; 返回值类型为 32 位整数
    ; 定义一个局部变量 cast210，用于存放后续的常量表达式结果值
    ; 该常量表达式用于获取全局字符串常量的地址
    %cast210 = getelementptr [13 x i8], [13 x i8]* @.str, i64 0, i64 0

    ; 将该字符串输出到标准输出流中
    call i32 @puts(i8* %cast210)
    ret i32 0
}

; 元数据定义，主要用来为优化器提供额外的信息，以便优化源代码结构
; 未命名元数据
!0 = !{i32 42, null, !"string"}
; 命名元数据
!foo = !{!0}
```

从上面的代码来看，整个 LLVM-IR 类似于一个低层次的 RISC 虚拟指令集。它支持加、减、乘、除及位移运算等简单的线性指令操作，而且我们还可以通过使用特定的修饰符来修饰变量、常量、函数等数据结构，来控制它们在应用程序运行时的具体表现形式。比如在上面的代码中，我们使用 `unnamed_addr` 修饰符来修饰 “@.str” 常量数据，该修饰符会告知优化器被修饰的常量在内存中的具体存放地址并不会被使用。因此，当该常量与其他常量具有相同值时，优化器便会将这些常量的存储地址进行统一合并，并且只在内存中保留一份资源副本，以节省内存资

源。实际上，是否需要为源代码中的数据结构附加各类属性修饰符，或者如何进行优化，都是由 LLVM 优化器来决定的，开发者并不需要考虑。

另外，与 RISC 指令集不同的是，在 LLVM-IR 体系中存在一套完整、简单的类型系统，每一个 IR 指令在实际执行时都需要强制指出操作数的具体类型。与 WebAssembly 中的类型构造符一样，这里的 i32 表示一个 32 位长的整数，f64 则表示一个 64 位的单精度浮点数。不仅如此，LLVM-IR 与单纯的机器码相比，还有一个明显不同的地方，就是它并不直接使用基于物理体系架构的实际寄存器名。取而代之的是，LLVM 假设在其构建的 IR 虚拟机环境中存在“无穷无尽”的类型寄存器，我们可以直接通过定义本地（%）或全局（@）标识符的方式来使用它们，而与具体的寄存器映射关系则完全交由特定的 LLVM 编译器后端来进行处理。

总的来看，在 LLVM 编译器描述的三段式结构中，其中间表示代码成为连接编译器前端与后端的桥梁。LLVM-IR 本身并不受特定编译器前端源语言类型的限制，包括 DSL 在内的任何形式的图灵完备或非完备的高级语言，均可以通过 LLVM 工具链提供的相应代码生成器或基于 LLVM 构建的编译器前端，根据源语言的语法特征将其转换为 LLVM-IR 代码格式。不仅如此，LLVM-IR 的加入和统一使用，使得 LLVM 项目能够把更多的精力放在如何对 LLVM-IR 代码进行性能优化上。LLVM 工具链在其开发套件中提供了多种可以被直接使用的优化器组件，这些优化器组件通过类似“管道”的处理方式被连续地应用在各类 LLVM-IR 代码上。经过优化器处理后的中间表示代码，也更有利于其在编译器后端进行使用。

4.1.3 基于 LLVM 的编译器架构

在基于 LLVM 构建的编译器链路中，编译器前端负责验证和解析源代码，并生成相应的 LLVM-IR 代码。当然，在生成特定的 LLVM-IR 代码前，编译器可能会先将源代码初步转换为 AST 结构。随后这些中间表示代码会被送往位于链路中间的 LLVM 优化器中进行优化处理。优化器会以管道的形式通过一系列分析和优化手段来改善代码质量，接下来这些优化后的中间代码便被送往符合特定体系架构的编译器后端以生成最终的本地机器码。如图 4-2 所示为基于 LLVM 组成的编译器链路结构。

与 GCC 等传统编译器一样，LLVM 工具链也提供了 llc（静态编译器）、llvm-link（链接器）等可以直接在命令行中通过手动调用来执行的“独立式”编译工具。除此之外，LLVM 还可以作为一套包含有完整编译器链路解决方案的通用代码组件库，可以方便地在各类应用中直接使用。这也是 LLVM 最重要的一个特性。如下是 LLVM 工具链已经封装好的一个全局优化器的部分 C++ 代码，通过这段代码我们可以了解到当 LLVM 被作为组件库使用时的基本方式。

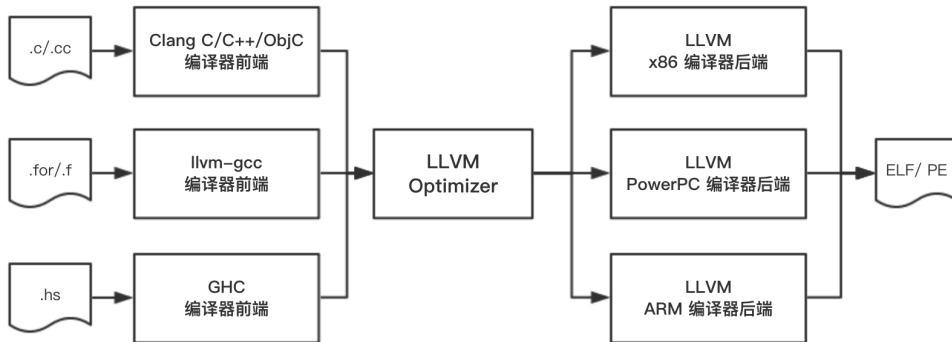


图4-2 基于LLVM组成的编译器链路结构

```

// 匿名命名空间
namespace {
// 优化器的实现类继承自 FunctionPass 类
class InstructionCombiningPass : public llvm::FunctionPass {
    InstCombineWorklist Worklist;
    const bool ExpensiveCombines;

public:
    static char ID; // 优化器标识符
    // 构造函数
    InstructionCombiningPass(bool ExpensiveCombines = true) : FunctionPass(ID),
ExpensiveCombines(ExpensiveCombines) {
        initializeInstructionCombiningPassPass(*PassRegistry::getPassRegistry());
    }
    // 实现父类中的虚函数
    void getAnalysisUsage(AnalysisUsage &AU) const override;
    bool runOnFunction(Function &F) override;
};
}

// 外部函数，用于返回一个优化器子类实体
FunctionPass *createInstructionCombiningPass(bool ExpensiveCombines) {
    return new InstructionCombiningPass(ExpensiveCombines);
}

```

可以看到，每一个独立的优化器(Pass)实现类都必须继承自 LLVM 工具链中的 `FunctionPass` 类，并且需要在子类中实现父类中定义的部分虚函数。包裹在子类外部的匿名命名空间主要用于对外隔离子类的具体实现细节，因此对于自定义的优化器，我们一般会将其实现在独立的 C++

源文件中。在整个优化器的源代码编写过程中，我们需要在该子类的内部实现父类中定义的 `runOnFunction` 等标准虚函数接口。另外，为了让该优化器能够被外部代码正常地调用，我们还需要在命名空间外提供一个可以用于返回该子类对应实例的方法。比如在上面的代码片段中，名为 `createInstructionCombiningPass` 的函数就是专门用于向外部调用者返回一个优化器类实例的。

在上面代码中定义的 `InstructionCombiningPass` 优化器会对传入其中的 LLVM-IR 代码进行“窥孔优化”。类似这样的优化器组件在整个 LLVM 工具链中被定义了多达几十个，并且它们都已经被完美地封装成独立的类结构。比如在基于 LLVM 构建的 Clang 编译器中，当我们在编译源代码时，可以为编译器指定对代码的优化等级参数。其中“-O0”表示不对源代码进行任何优化，当指定该参数后，编译器不会使用 LLVM 工具链提供的任何优化器组件；而当指定“-O3”优化参数后，Clang 则会使用 LLVM 工具链提供的众多优化器组件，来对源代码依次进行相应的优化处理。

4.1.4 LLVM 优化策略

本节我们将介绍在 LLVM 优化器中经常使用到的几种简单的 IR 代码优化策略。这些优化策略不仅可以被应用在以 LLVM-IR 为基础的中间表示代码上，而且对于传统的编译器开发流程，也可以将其应用在任意一个阶段的中间表示代码上。对于虚拟机体系来说，这些优化策略可以直接应用在诸如 Wasm 模块其内部的虚拟机二进制 `OpCode` 这类专用指令代码上。从另一个方面来看，这些优化策略本质上与具体的源语言类型无关。

死码消除优化

在编译器理论中，所谓的死码消除（Dead Code Elimination, DCE）其实是一种常用的编译器优化技术，我们在第 1 章的实践内容中也介绍过。该技术主要通过删除那些不会影响应用程序运行结果的代码来优化应用程序占用的存储空间大小并提高其运行效率，并且该优化技术还可以在一定程度上简化应用程序的代码结构，为接下来的进一步代码优化做好准备。通常来说，源代码中的死码主要分为两种：第一种是永远无法被执行到的代码；第二种是仅用于保存数据，但没有被实际使用到的变量代码。现在我们对如下 C/C++ 代码进行 DCE 优化。

```
int add (int a, int b) {  
    int x = 10;  
    // [DCE]: 未使用的变量  
    int y = 11;  
  
    if (x > 0) {
```

```

    return a + b;
}
// [DCE]: 无法执行的代码
y = 12;
return 2 * (a + b);
}

```

首先，优化器会分析上述代码中 `add` 函数内各个变量的作用域，以及对应变量值的使用情况。这里由于“`y`”被声明为函数内部的一个局部变量，并且该变量的值并没有在函数内任意语句中使用，因此其所在的定义语句会被优化器标识为目标消除代码。接下来，在函数的第一个条件控制结构中，`return` 语句的执行条件“`x>0`”永远成立，因此在所有情况下该语句都会被执行，这将导致操作系统的指令指针提前从该函数的上下文中退出，使得在该语句后面出现的所有代码均无法被正常执行。所以，优化器也会将在该条件语句结构后出现的所有代码均标记为目标消除代码，待分析结束后统一从优化结果中进行消除。

通常来说，DCE 优化技术的另一个常见的使用场景，便是通过结合 C/C++ 的预处理器特性来完成对源代码的选择性编译过程。比如在下面的代码段中，我们通过在源代码文件头部或对应头文件中定义的若干个预处理器变量来控制整个应用程序的实际编译结果。可以看到，在主函数 `main` 中，通过使用条件选择结构，代码根据预处理器变量 `ARCH` 的不同取值来为变量“`x`”赋予不同的值。我们知道，源代码在编译阶段并不会进行预处理操作，而且预处理过程也只是在代码编译阶段前编译器对源代码中特定的标识符进行字面量上的文本替换，而这些标记则会在编译过程中被再次解析为对应的 C/C++ 字面量数据值。因此在代码编译过程中，编译器会发现条件选择结构中的条件取值永远为真。由于位于 `else` 分支中的代码永远不会被执行，因此在 IR 优化阶段优化器便会在这部分代码采用 DCE 优化策略进行优化处理。

```

#include <iostream>
// 定义预处理器
#define ARCH "X86"

int main (int argc, char *argv[]) {
    int x;
    // 基于 DCE 的条件编译
    if (ARCH == "X86") { // 编译时为 if ("X86" == "X86") { ...
        x = 10;
    } else {
        x = 0;
    }
    std::cout << x << std::endl;
}

```

```
    return 0;
}
```

当然，我们也可以使用如下代码所示的传统预编译处理方式，来实现对代码的选择编译过程。具体使用哪种方式来进行条件编译，对编译器最后生成的二进制文件本身并没有任何影响，我们可以根据项目规定的开发规范，以及目标编译器所支持的功能特性来进行选择。相对来说，减少编译器进行 DCE 过程所花费的时间，可以在一定程度上提高源代码的整体编译效率。

```
#include <iostream>
// 定义预处理器
#define THIS_X86_ARCH

int main (int argc, char *argv[]) {
    int x;
    // 基于预处理器的条件编译
    # ifdef THIS_X86_ARCH
        x = 10;
    # else
        x = 0;
    std::cout << x << std::endl;
    return 0;
}
```

常量折叠/传播优化

常量折叠/传播（Constant Folding/Propagation）也是编译器常用的一种代码优化技术。通过该优化策略，编译器可以将源代码中的常量表达式字面量化，从而删除不必要的表达式运行时计算过程。比如下面给出的这段 C/C++ 代码。

```
#include <iostream>

using namespace std;

int main(int argc, char **argv) {
    int i = 100 * 80 + 10 * 10; // 如果指定了 constexpr 关键字，则可帮助编译器识别常量表达式
    cout << i << endl;

    return 0;
}
```

可以看到，我们在 main 主函数中定义了一个同时含有加法和乘法运算的常量表达式，并将

该表达式的最终计算结果赋值给了整型变量“i”。实际上，大多数现代编译器在编译这段代码时，并不会为其中的常量表达式生成对应的加法和乘法指令。相反的，编译器会识别出常量表达式的结构，并在编译时将其直接替换为经过计算后的表达式结果值。比如上面代码中的变量“i”，在实际编译时便会被编译器直接优化成一个存储了整数值“900”的整型变量。这种将源代码中常量表达式直接替换为其最终计算结果值的优化策略，便被称为“常量折叠”。

从另一方面来看，既然编译器可以将引用了常量表达式的变量，通过优化手段将其转换为具有单一值引用的普通变量，那么也可以对源代码中所有引用了常量表达式的变量进行提前求值，以优化最终应用程序对应机器码中的运算指令数量。这种优化策略被称为“常量传播”。比如对于下面这段 C/C++ 源代码，编译器首先会将源代码中所有使用到变量“i”的常量表达式中的该变量直接以其值“10”进行替换。替换后变量“j”变成了一个只带有纯字面量数值计算的表达式，此时优化器会继续通过常量折叠策略对使用到变量“j”的常量表达式进行优化。

```
#include <iostream>

using namespace std;

int main(int argc, char **argv) {
    int i = 10;
    int j = i + 10;
    int k = j * 2 + 100;
    cout << k << endl;

    return 0;
}
```

总的来说，常量折叠和常量传播这两种优化策略的主要思路都是从源代码中尽可能多地删除不必要的静态代码，将一些可以提前求值的变量用对应的表达式结果以常量值的形式进行替换。实际上，通常编译器会将 DCE、常量折叠和常量传播这三种优化策略联合起来使用。比如对于下面这段 C/C++ 代码（案例来自维基百科）。

```
int a = 30;
int b = 9 - (a / 5);
int c;

c = b * 4;
if (c > 10) {
    c = c - 10;
```

```
}  
return c * (60 / a);
```

首先，编译器会通过常量传播策略对这段代码进行优化。这里编译器会将代码中所有常量表达式使用到变量“a”的地方都以该变量的值“30”来进行替换。优化后的代码如下。

```
int a = 30;  
int b = 9 - (30 / 5);  
int c;  
  
c = b * 4;  
if (c > 10) {  
    c = c - 10;  
}  
return c * (60 / a);
```

然后，通过常量折叠优化策略，编译器会将变量“b”对应的常量表达式以其最终计算值来进行替换。处理后的代码如下。

```
int a = 30;  
int b = 3;  
int c;  
  
c = b * 4;  
if (c > 10) {  
    c = c - 10;  
}  
return c * (60 / a);
```

继续重复执行这两种代码优化策略两次后，我们可以得到如下所示的代码。可以看到，在这段代码中已经不存在常量表达式了。需要注意的是，代码中变量“c”的值在常量传播的优化过程中并没有被完全替换。这是由于代码中存在的条件选择结构会直接影响后续常量表达式中变量“c”的实际值，因此编译器只能够替换部分可确定的变量值。

```
int a = 30;  
int b = 3;  
int c;  
  
c = 12;  
if (true) {  
    c = 2;  
}  
return c * 2;
```


接下来，编译器继续通过 DCE 策略来进一步优化代码，优化后的代码如下。

```
int c;  
c = 12;  
if (true) {  
    c = 2;  
}  
return c * 2;
```

至此，采用以上三种优化策略进行的代码优化过程只能达到这样的程度。可能读者已经发现，上面这段代码还有很大的优化空间，比如还可以对其中已经被确定恒为“真”的条件选择结构进行优化。

为了进一步优化代码，我们需要为编译器补充一种策略——稀疏条件常量优化。通过该策略，编译器可以对代码中恒成立的条件选择结构进行优化，优化后的代码如下。

```
return 4;
```

需要注意的是，实际上编译器并不会直接对原始的语言源代码进行上述优化。这里只是为了能够直观地展现优化策略对源代码组成结构的影响，而在源代码上直接体现出了优化策略的效果。相对的，对源代码对应的中间表示代码进行优化会更加简单和高效。这也是大部分现代编译器的代码优化目标，比如基于 LLVM 优化器与 LLVM-IR 构建的 Clang 编译器。

4.1.5 LLVM 命令行工具

本节我们将了解在 LLVM 工具链套件内提供的一系列非常实用的命令行工具。通过这些工具，我们可以方便地在本地对基于 LLVM 开发的各类项目、源代码和 LLVM-IR 代码进行调试与分析。

Clang 编译器——clang

从本质上讲，Clang 编译器并不属于 LLVM 命令行工具集的一部分，但是由于其基于 LLVM 工具链开发的特点，使得它拥有很多其他类似编译器所不具备的功能特性。接下来将要介绍的便是 Clang 编译器具有的可以将高级语言源代码转译为 LLVM-IR 中间代码的特性。比如对于如下这段 C/C++ 源代码。

```
llvm-ir.cc  
int main() {  
    int a = 30;  
    int b = 9 - (a / 5);
```

```

int c;

c = b * 4;
if (c > 10) {
    c = c - 10;
}

return c * (60 / a);
}

```

我们通过如下命令将这段源代码转译为对应的 LLVM-IR 代码。

```
clang -S -emit-llvm llvm-ir.cc
```

在上面的命令语句中，参数“-S”用于指定编译器生成包含有可读性汇编代码的目标文件；参数“-emit-llvm”用于设置编译器以 LLVR-IR 的形式来生成目标文件，该参数需要配合“-S”参数一起使用。当该命令语句执行完毕后，编译器会在当前文件夹内生成一个名为“llvm-ir.ll”的文件，文件名后缀“.ll”便表示这是一个包含有可读 LLVM-IR 代码的 ASCII 文本文件。如图 4-3 所示为该文件内主函数部分对应的 LLVM-IR 代码。可以看到，由于没有使用任何代码优化策略，在这段简单的计算逻辑中，LLVM-IR 一共使用了多达 19 个寄存器（代码中以“%”标识的本地寄存器）来完成相应的数学运算，以及数据打印过程。

```

; Function Attrs: noinline norecurse nounwind optnone ssp uwtable
define i32 @main() #0 {
    %1 = alloca i32, align 4
    %2 = alloca i32, align 4
    %3 = alloca i32, align 4
    %4 = alloca i32, align 4
    store i32 0, i32* %1, align 4
    store i32 30, i32* %2, align 4
    %5 = load i32, i32* %2, align 4
    %6 = sdiv i32 %5, 5
    %7 = sub nsw i32 9, %6
    store i32 %7, i32* %3, align 4
    %8 = load i32, i32* %3, align 4
    %9 = mul nsw i32 %8, 4
    store i32 %9, i32* %4, align 4
    %10 = load i32, i32* %4, align 4
    %11 = icmp sgt i32 %10, 10
    br i1 %11, label %12, label %15

; <label>:12:                                     ; preds = %0
    %13 = load i32, i32* %4, align 4
    %14 = sub nsw i32 %13, 10
    store i32 %14, i32* %4, align 4
    br label %15

; <label>:15:                                     ; preds = %12, %0
    %16 = load i32, i32* %4, align 4
    %17 = load i32, i32* %2, align 4
    %18 = sdiv i32 60, %17
    %19 = mul nsw i32 %16, %18
    ret i32 %19
}

```

图4-3 上述C/C++源代码中主函数对应的LLVM-IR代码

我们可以尝试在编译命令语句中为 Clang 添加一个“-O3”级别的优化参数。经过优化后生成的 LLVM-IR 代码如图 4-4 所示。这与我们之前在“LLVM 优化策略”示例中得到的最终优化结果是一致的。

```
; Function Attrs: norecurse nounwind readnone ssp uwtable
define i32 @main() local_unnamed_addr #0 {
    ret i32 4
}
```

图4-4 经过LLVM优化器处理后的LLVM-IR代码

LLVM-IR 解释器——lli

通过 lli 命令，我们可以直接调用专门为 LLVM-IR 设计的即时解释器，或称为“动态编译器”。解释器会直接逐行解释并执行目标文件内的 IR 代码。这里我们在上面代码的基础上添加了用于打印计算结果的代码段，以方便观察 lli 命令的执行效果。修改后的 C/C++源代码如下。

```
llvm-lli.cc
#include <iostream>
int main() {
    int a = 30;
    int b = 9 - (a / 5);
    int c;

    c = b * 4;
    if (c > 10) {
        c = c - 10;
    }

    std::cout << "The compute result is: " << c * (60 / a) << std::endl;
    return 0;
}
```

现在重新编译源文件，并直接在本地通过 lli 命令来解释执行生成的包含有 LLVM-IR 代码的.ll 文件，如图 4-5 所示。

```
→ wasm clang -S -emit-llvm -O3 llvm-lli.cc
→ wasm ./bin/lli llvm-lli.ll
The compute result is: 4
```

图4-5 编译并通过lli命令解释执行生成的LLVM-IR代码

LLVM-IR 优化器——opt

通过 opt 命令，我们可以直接在命令行中调用 LLVM 工具链提供的 IR 代码优化器来对

LLVM-IR 代码进行优化。该优化器同时支持对可读文本及二进制格式下的 LLVM-IR 代码进行优化，并且可以通过添加参数来手动指定相应的优化策略。接下来，我们将通过逐渐叠加优化策略的方式，来对下面这段 C/C++ 源代码对应的 LLVM-IR 代码逐步进行优化。

```
int main() {
    int a = 30;
    int b = 100;
    for (int i = 0; i < 10; i++) {
        a += i;
    }
    return b * 10 + 100;
}
```

首先，通过 Clang 编译器将这段源代码编译成对应的 LLVM-IR 代码。在生成的代码文件中，主函数对应的 LLVM-IR 代码如图 4-6 所示。

```
; Function Attrs: noinline norecurse nounwind optnone ssp uwtable
define i32 @main() #0 {
    %1 = alloca i32, align 4
    %2 = alloca i32, align 4
    %3 = alloca i32, align 4
    %4 = alloca i32, align 4
    store i32 0, i32* %1, align 4
    store i32 30, i32* %2, align 4
    store i32 100, i32* %3, align 4
    store i32 0, i32* %4, align 4
    br label %5

; <label>:5:                                     ; preds = %12, %0
    %6 = load i32, i32* %4, align 4
    %7 = icmp slt i32 %6, 10
    br i1 %7, label %8, label %15

; <label>:8:                                     ; preds = %5
    %9 = load i32, i32* %4, align 4
    %10 = load i32, i32* %2, align 4
    %11 = add nsw i32 %10, %9
    store i32 %11, i32* %2, align 4
    br label %12

; <label>:12:                                    ; preds = %8
    %13 = load i32, i32* %4, align 4
    %14 = add nsw i32 %13, 1
    store i32 %14, i32* %4, align 4
    br label %5

; <label>:15:                                    ; preds = %5
    %16 = load i32, i32* %3, align 4
    %17 = mul nsw i32 %16, 10
    %18 = add nsw i32 %17, 100
    ret i32 %18
}
```

图4-6 上述C/C++源代码中主函数对应的LLVM-IR代码

可以看到，在这段 IR 代码中包含有很多可以被去掉的无效冗余代码。比如通过多个 <label> 标签模拟出的 for 循环结构，以及并没有被实际使用到的名为“a”的变量。接下来，我们将通过 opt 命令调用 LLVM 优化器来对上述代码进行优化。首先为优化器添加“-mem2reg”优化策

略，该策略会将 IR 内的内存级变量引用提升为寄存器级变量引用。优化器的调用命令如下。

```
./bin/opt -S -mem2reg llvm-opt.ll
```

在该命令中，参数“-S”表示直接返回并输出经过优化器优化后的 LLVM-IR 代码，如图 4-7 所示。可以看到，在原有代码中，基于 load 与 store 指令进行的内存级变量引用已经全部变为基于本地寄存器(%)形式的变量值引用，并且代码的整体行数也有所减少。下面我们继续为优化器添加“-constprop”与“-dce”优化策略，这两个策略分别对应于“常量传播”与“死码消除”优化。

```
; Function Attrs: norecurse nounwind ssp uwtable
define i32 @main() #0 {
    br label %1

; <label>:1:                                     ; preds = %5, %0
    %01 = phi i32 [ 30, %0 ], [ %4, %5 ]
    %0 = phi i32 [ 0, %0 ], [ %6, %5 ]
    %2 = icmp slt i32 %0, 10
    br i1 %2, label %3, label %7

; <label>:3:                                     ; preds = %1
    %4 = add nsw i32 %01, %0
    br label %5

; <label>:5:                                     ; preds = %3
    %6 = add nsw i32 %0, 1
    br label %1

; <label>:7:                                     ; preds = %1
    %8 = mul nsw i32 100, 10
    %9 = add nsw i32 %8, 100
    ret i32 %9
}
```

图4-7 经过“-mem2reg”优化策略处理后的LLVM-IR代码

这里直接将优化策略名称当作参数传递给 opt 命令即可。在命令行中执行如下命令语句。

```
./bin/opt -S -mem2reg -constprop -dce llvm-opt.ll
```

在命令输出结果中，经过三种优化策略处理后的 LLVM-IR 代码如图 4-8 所示。

```
; Function Attrs: norecurse nounwind ssp uwtable
define i32 @main() #0 {
    br label %1

; <label>:1:                                     ; preds = %5, %0
    %01 = phi i32 [ 30, %0 ], [ %4, %5 ]
    %0 = phi i32 [ 0, %0 ], [ %6, %5 ]
    %2 = icmp slt i32 %0, 10
    br i1 %2, label %3, label %7

; <label>:3:                                     ; preds = %1
    %4 = add nsw i32 %01, %0
    br label %5

; <label>:5:                                     ; preds = %3
    %6 = add nsw i32 %0, 1
    br label %1

; <label>:7:                                     ; preds = %1
    ret i32 1100
}
```

图4-8 经过三种优化策略处理后的LLVM-IR代码

可以看到，位于“<label>:7”标签内的代码已经被优化成最简单的形式，即直接从函数中返回常量表达式的计算终值（1100）。最后，还需要将这段代码中多余的循环结构移除。下面我们继续为优化器添加“-loop-deletion”优化策略，该策略主要用于移除代码中的无效循环体结构。经过4种优化策略连续优化处理后，可以得到如图4-9所示的最优代码。

```

; Function Attrs: norecurse nounwind ssp uwtable
define i32 @main() #0 {
    br label %1

; <label>:1:                                     ; preds = %0
    ret i32 1100
}

```

图4-9 经过4种优化策略处理后的LLVM-IR代码

LLVM 静态编译器——llc

“llc”是 LLVM 命令行工具集提供的一个静态编译器。通过该编译器，可以将一个包含有 LLVM-IR 代码的“.ll”文件编译成以“.s”为后缀的基于特定平台架构的汇编代码文件。我们可以通过如下命令，将上面经过多次优化后得到的 LLVM-IR 代码编译成包含有当前平台架构指令的汇编代码。

```
./bin/llc llvm-opt.ll
```

该命令执行结束后，会在当前文件夹内生成一个以“.s”为后缀且与源文件同名的 ASCII 文本文件，如图4-10所示为该文件的内容。

```

.section      __TEXT,__text,regular,pure_instructions
.macrosx_version_min 10, 13
.globl _main          ## -- Begin function main
.p2align      4, 0x90
_main:
.cfi_startproc
## %bb.0:
    pushq    %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq     %rsp, %rbp
    .cfi_def_cfa_register %rbp
    movl     $1100, %eax    ## imm = 0x44C
    popq     %rbp
    retq
    .cfi_endproc
                                ## -- End function
.subsections_via_symbols

```

图4-10 “.s”汇编代码文件的内容

LLVM 汇编器——llvm-as

通过 llvm-as 命令行工具，可以将包含有可读文本格式的 LLVM-IR 文件转换为二进制格式的 LLVM 比特码。比如这里通过如下命令将之前由 Clang 编译器生成的“.ll”文件编译为对应

的“.bc”文件。

```
./bin/llvm-as llvm-opt.ll
```

该命令执行完毕后，会在当前目录下生成同名的以“.bc”为后缀的文件。我们可以通过 hexdump 命令来查看该二进制文件的具体内容，如图 4-11 所示。

```
→ wasm ./bin/llvm-as llvm-opt.ll
→ wasm hexdump -C llvm-opt.bc
00000000 de c0 17 0b 00 00 00 00 14 00 00 00 14 08 00 00 |.....|
00000010 07 00 00 01 42 43 c0 de 35 14 00 00 05 00 00 00 |...BC..5.....|
00000020 62 0c 30 24 49 59 be a6 ee d3 3e 2d 44 01 32 05 |b.0$IY...>-D.2.|
00000030 00 00 00 00 21 0c 00 00 c7 01 00 00 0b 02 21 00 |.....!.....|
00000040 02 00 00 00 13 00 00 00 07 81 23 91 41 c8 04 49 |.....#.A..I|
00000050 06 10 32 39 92 01 84 0c 25 05 08 19 1e 04 8b 62 |..29...%......b|
00000060 80 10 45 02 42 92 0b 42 84 10 32 14 38 08 18 4b |..E.B..B..2.8..K|
00000070 0a 32 42 88 48 90 14 20 43 46 88 a5 00 19 32 42 |.2B.H.. CF...2B|
00000080 04 49 0e 90 11 22 c4 50 41 51 81 8c e1 83 e5 8a |.I...".PAQ.....|
00000090 04 21 46 06 51 18 00 00 9b 00 00 00 1b de 26 f8 |.!F.Q.....&.|
000000a0 ff ff ff ff 01 80 03 40 02 34 20 0c 88 71 78 07 |.....@.4 ..qx.|
000000b0 79 90 87 72 18 07 7a 60 87 7c 68 03 79 78 87 7a |v...r...z...|
```

图4-11 “.bc”比特码文件的部分内容

LLVM 符号表查看器——llvm-nm

通过 llvm-nm 命令行工具，我们可以查看一个包含有二进制 LLVM-IR 比特码的“.bc”文件其内部的符号表信息。这里还是以上面生成的“.bc”文件为例，命令如下。

```
./bin/llvm-nm -aA llvm-opt.bc
```

在上述命令语句中，“-a”参数用于指定 llvm-nm 显示包括调试器私有符号在内的所有符号表符号；“-A”参数用于在输出结果中显示每个符号的来源文件名。如图 4-12 所示为该命令的执行结果。可以看到，在这个 LLVM 模块中只存在一个名为“_main”的符号，该符号对应着 C/C++ 源代码中的主函数。在输出结果中，“T”标识符表示该符号的具体类型，这里表示一个全局对象函数。后面紧接着的是该符号的具体名称。关于其他类型说明符的具体介绍，可以参考 LLVM 官方文档进行了解。

```
→ wasm ./bin/llvm-nm -aA llvm-opt.bc
llvm-opt.bc: ----- T _main
```

图4-12 “.bc”比特码文件内的符号表信息

编译和运行

在上面几个小节中，我们生成了多个包含有不同状态 LLVM-IR 中间代码，甚至是面向特定平台架构底层汇编代码的文件(.ll/.bc/.s)。对于这些文件，我们都可以在最后通过 Clang 编译器将它们编译成对应的二进制可执行文件。类似的命令如下。

```
clang -O3 llvm-opt.ll -o llvm-opt
```

4.1.6 WebAssembly 与 LLVM

在本书的第 1 章内容中，我们介绍过将 C/C++ 源代码编译成二进制 Wasm 模块的几种方式。其中第一种方式是首先通过 Emscripten 工具链中基于 Clang 编译器构建的 emcc 编译器前端，将 C/C++ 源代码编译成 LLVM-IR 中间比特码的形式。然后再通过基于 LLVM 构建的 Fastcomp 编译器后端，将这些 IR 代码编译成 ASM.js 形式的 JavaScript 子集代码。最后，还需要借助 Binaryen 工具链提供的 asm2wasm 工具，将这些 ASM.js 代码编译成 Wasm 二进制文件。这种方式的具体操作流程如图 4-13 所示。



图4-13 通过emcc/Fastcomp和asm2wasm构建Wasm模块的编译链路

第二种方式则分为 4 个步骤来进行。首先，直接通过 Clang 编译器将含有 C/C++ 源代码的文件编译成 LLVM-IR 中间代码，这里对 LLVM-IR 的具体表现状态不做要求，可读文本代码或二进制比特码的形式均可。然后，通过 LLVM 命令行工具集中的 llc 静态编译器，将这些中间表示代码编译成与平台相关的汇编代码。需要注意的是，这里我们说的平台是指 WebAssembly 这个虚拟的指令集平台，而不是本地操作系统运行所在的平台。最后，再借助 Binaryen 工具链提供的 s2wasm 命令行工具，将这些汇编代码编译成二进制格式的 Wasm 模块。这种方式的具体操作流程如图 4-14 所示。

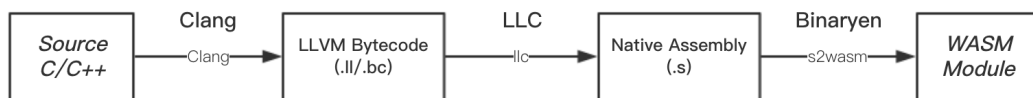


图4-14 通过clang、llc和s2wasm构建Wasm模块的编译链路

在上面两种生成 Wasm 二进制模块的具体方式中，我们多次看到 LLVM 的身影。由于第一种方式是基于 Emscripten 工具链进行的，因此相关内容留到第 5 章中再进行讨论。对于第二种方式，我们将以如下 C/C++ 源代码作为例子来观察该流程中的一些重要细节。

```

fibonacci.cc
#ifdef __cplusplus
extern "C" {
#endif
    int fib (int x) {
        if (x < 2) {

```



```

    return 1;
} else {
    return fib(x - 1) + fib(x - 2);
}
}
#endif __cplusplus
}
#endif

```

按照前面总结的步骤，首先通过 Clang 编译器将这段源代码编译成 LLVM-IR 中间代码。对应的命令如下。

```
clang -O3 -S -emit-llvm fibonacci.cc
```

该命令执行完毕后，会在目标目录下生成同名的以“.ll”为后缀的 ASCII 文件，该文件中的部分内容如图 4-15 所示。

```

; ModuleID = 'fibonacci.cc'
source_filename = "fibonacci.cc"
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.12.0"

; Function Attrs: nounwind readnone ssp uwtable
define i32 @fib(i32) local_unnamed_addr #0 {
    %2 = icmp slt i32 %0, 2
    br i1 %2, label %13, label %3

; <label>:3:                                ; preds = %1
    br label %4

; <label>:4:                                ; preds = %3, %4
    %5 = phi i32 [ %9, %4 ], [ %0, %3 ]
    %6 = phi i32 [ %10, %4 ], [ 1, %3 ]
    %7 = add nsw i32 %5, -1
    %8 = tail call i32 @fib(i32 %7)
    %9 = add nsw i32 %5, -2
    %10 = add nsw i32 %8, %6
    %11 = icmp slt i32 %9, 2
    br i1 %11, label %12, label %4

; <label>:12:                               ; preds = %4
    br label %13

```

图4-15 “.ll”文件中的部分内容

然后，通过 LLVM 命令行工具集中的“llc”静态编译器，将上述“.ll”文件编译成基于 WebAssembly 虚拟平台的汇编文件，命令如下。这里 llc 工具会直接使用在 LLVM 工具链内部专门为 WebAssembly 平台设计的实验性编译器后端，来将 LLVM-IR 代码转译成 GAS 形式的汇编指令代码。这一步骤是所有基于 LLVM 构建的编译器后端所必须经历的。

```

./bin/llc fibonacci.ll
-mtriple=wasm32-unknown-unknown-elf
-filetype=asm
-o fibonacci.s

```

可以看到，在上面的命令中，通过“-mtriple”参数为 llc 工具指定了生成汇编文件时需要参考的目标平台类型，参数值“wasm32-unknown-unknown-elf”正对应于 WebAssembly 虚拟平台的 Triple 字符串。所谓的 Triple 字符串，是指在 LLVM 内部用于表示某一具体平台架构的描述字符串，该字符串可以由 3~4 个具有明确含义的字段组成，每个字段都代表该平台架构不同方面的相关信息。比如对于一个包含有 4 个字段的 Triple 字符串，如果按照字段连接符从左至右的顺序来看，各字段的含义可以被描述为“目标平台架构类型—软/硬件供应商—操作系统类型—附加的环境参数”。这里由于 WebAssembly 属于一种虚拟的硬件架构类型，并不依赖特定的硬件供应商及操作系统类型，因此将对应的字段均设置为“unknown”。

需要注意的是，由于当前 WebAssembly 还没有被 LLVM 加入编译时默认支持的目标平台列表中，因此在编译 LLVM 工具链时，需要通过额外指定一个“-DLLVM_EXPERIMENTAL_TARGETS_TO_BUILD=WebAssembly”参数来将 WebAssembly 加入编译目标平台中。在编译完成后，我们可以通过查看 llc 工具的版本信息，来打印出当前编译的 LLVM 工具链支持的全部目标平台类型，详情如图 4-16 所示。

```
→ wasm ./bin/llc -version
LLVM (http://llvm.org/):
LLVM version 6.0.0
DEBUG build with assertions.
Default target: x86_64-apple-darwin16.6.0
Host CPU: broadwell

Registered Targets:
aarch64      - AArch64 (little endian)
aarch64_be   - AArch64 (big endian)
amdgc8       - AMD GCN GPUs
arm          - ARM
arm64        - ARMv8 (little endian)
armeb        - ARM (big endian)
bpf          - BPF (host endian)
bpfel        - BPF (big endian)
bpfel        - BPF (little endian)
hexagon      - Hexagon
lanai        - Lanai
mips         - Mips
mips64       - Mips64 [experimental]
mips64el     - Mips64el [experimental]
mipsel       - Mipsel
msp430       - MSP430 [experimental]
nvptx        - NVIDIA PTX 32-bit
nvptx64      - NVIDIA PTX 64-bit
ppc32        - PowerPC 32
ppc64        - PowerPC 64
ppc64le      - PowerPC 64 LE
r600         - AMD GPUs HD2XXX-HD6XXX
sparc        - Sparc
sparcel      - Sparc LE
sparcv9      - Sparc V9
systemz      - SystemZ
thumb        - Thumb
thumbel      - Thumb (big endian)
wasm32       - WebAssembly 32-bit
wasm64       - WebAssembly 64-bit
x86          - 32-bit X86: Pentium-Pro and above
x86-64       - 64-bit X86: EM64T and AMD64
xcore        - XCore
```

图4-16 当前LLVM工具链支持的全部目标平台类型

可以看到，在 `llc` 工具的版本信息中列出了 `wasm32` 和 `wasm64` 两个 WebAssembly 的目标平台类型。当上面的命令执行完毕后，我们可以在目标文件夹内找到对应的包含有可读汇编代码的“.s”格式文件。如果此时查看该文件的内容，则会发现所谓的汇编代码其实就是由众多的 WebAssembly 虚拟指令组成的。如图 4-17 所示，这里有我们熟悉的用于定义 32 位整数类型常量的 `i32.const`，以及用于计算两个 32 位整数之和的 `i32.add` 等各类虚拟指令对应的可读文本代码。不仅如此，汇编代码中的 `push`、`pop` 等指令也证实了 WebAssembly 虚拟机所具有的堆栈机模型结构。

```
.text
.file "fibonacci.cc"
.globl fib
.type fib,@function

fib:
    .param          i32
    .result         i32
    .local          i32

# %bb.0:
    i32.const       $l=, 1
    block
    i32.const       $push0=, 2
    i32.lt_s        $push1=, $0, $pop0
    br_if           0, $pop1      # 0: down to label0

# %bb.1:
    i32.const       $l=, 1
.LBB0_2:
    loop
    i32.const       $push7=, -1
    i32.add          $push2=, $0, $pop7
    i32.call         $push3=, fib@FUNCTION, $pop2
    i32.add          $l=, $pop3, $l
    i32.const       $push6=, -2
```

图4-17 基于wasm32目标平台类型构建的“.s”文件部分内容

从某个角度来看，我们可以将 WebAssembly 虚拟机看作是一种特殊的物理目标平台类型，该目标平台具有以虚拟指令为主的特定指令集架构（Virtual-ISA）。而上面通过 `llc` 工具生成的“.s”汇编文件便是基于该平台的具体指令集及虚拟机架构模式生成的。实际上，从官方团队的角度来看，从 LLVM-IR 代码转译到汇编代码这一步实际上没有非常明确的作用，从某种程度上讲，只是在基于 LLVM 开发 Wasm 模块编译器后端时所必须经历的一个步骤而已。不仅如此，由于 Wasm 虚拟机本身不同于 MIPS 架构等各类传统 ISA 指令集指令及运行时特性，导致从 LLVM 后端生成的 WebAssembly 汇编代码缺少足够的灵活性，并且代码本身也并没有得到足够的优化。在现阶段，这些不足都将在 Binaryen 工具链中被弥补。接下来，我们将通过 Binaryen 工具链中的 `s2wasm` 工具将上述 LLVM 下的 Wasm 汇编代码编译成 WAT 格式的 Wasm 模块。

```
./binaryen/bin/s2wasm fibonacci.s > fibonacci.wat
```

通过上面的命令语句，我们可以将含有 Wasm 汇编代码的“.s”文件编译成含有可读文本代码的标准 Wasm 二进制模块。这里 `s2wasm` 工具首先会将 Wasm 汇编代码转译成 Binaryen 工具链特有的一种中间表示代码，这种代码相比于从 LLVM 后端输出的汇编代码更加贴近 Wasm 本身的语法和结构，同时也更易于在内存中进行优化处理。另外，相比于 LLVM 后端代码生成

器的多线程优化处理过程，Binaryen 可以采用多线程的方式来加快代码的优化及编译速度，同时占用更少的本地内存。最后，只需要再通过 Binaryen 工具链中的 wasm-as 工具将这些 Wasm 可读文本代码编译成二进制格式的 Wasm 模块即可。由于模块代码本身已经过优化，因此这一步的汇编过程将十分简单和快速。

```
./binaryen/bin/wasm-as fibonacci.wat -o fibonacci.wasm
```

该命令执行完毕后，会在当前目录下生成名为“fibonacci.wasm”的 Wasm 二进制模块文件。如图 4-18 所示，我们可以通过 hexdump 命令来查看该文件的二进制内容，位于模块开头部分的魔术字符“asm”证明了这是一个标准的 Wasm 模块。

```
→ wasm hexdump -C fibonacci.wasm
00000000 00 61 73 6d 01 00 00 00 01 86 80 80 80 00 01 60 |.asm.....|
00000010 01 7f 01 7f 03 82 80 80 80 00 01 00 04 84 80 80 |.....|
00000020 80 00 01 70 00 00 05 83 80 80 80 00 01 00 01 07 |...p.....|
00000030 90 80 80 80 00 02 06 6d 65 6d 6f 72 79 02 00 03 |.....memory...|
00000040 66 69 62 00 00 09 81 80 80 80 00 00 0a be 80 80 |fib.....|
00000050 80 00 01 b8 80 80 80 00 01 01 7f 02 7f 41 01 21 |.....A.!|
00000060 01 02 40 20 00 41 02 48 0d 00 41 01 21 01 03 40 |..@ .A.H..A!..@|
00000070 20 00 41 7f 6a 10 00 20 01 6a 21 01 20 00 41 7e |.A.j.. .j!. .A~|
00000080 6a 21 00 20 00 41 01 4a 0d 00 0b 0b 20 01 0b 0b |j!. .A.J....|
00000090
```

图4-18 借助Binaryen工具链编译生成Wasm模块

总的来说，WebAssembly 团队之所以会选择基于 LLVM 工具链来构建 Wasm 模块的编译器后端，一方面是因为从 Emscripten 工具链与 PNaCl 技术的开发过程中吸取了相应的经验和教训；另一方面也是由于 Wasm 技术在发展初期其目标语言主要以 C/C++ 为主，基于 LLVM 来构建编译器后端可以使 WebAssembly 直接复用各类已经十分成熟的源语言编译器前端，以及相应的代码优化器。不仅如此，由于 LLVM 本身已经流程化的编译器后端开发流程，也可以使团队成员把更多的精力放在诸如 Wasm 的新标准制定、技术细节及性能优化等更为重要的工作上。

4.2 基于 LLVM 定义新的编程语言

注：本节编译器实现的全部源代码可以在 <https://github.com/Becavalier/Cinderella> 对应的 Github 仓库中进行查看。

在本节中，我们将重新定义一种名为“Cinderella”的全新编程语言，并为其实现一个简单的、完整的编译器（前端+后端），同时以此来观察基于 LLVM 工具链构建编译器前端并整合优化器及编译器后端的整个流程。在这个过程中，我们将会介绍在开发编译器时需要使用到的相关技术，以及对应的代码实现。但限于篇幅，以及本书的主要内容，这里并不会采用标准的编译器前、后端开发流程来构建该编译器，而是采用更为基础和简单的方法来实现。本节的主要目标是让读者能够从整体上了解基于 LLVM 开发编译器的基本流程、将 LLVM 工具链作为

可重用 Library 组件库的具体使用方法，以及借助 LLVM 来构建编译器各个代码处理阶段的基本实现思路等内容。

在着手开发编译器之前，我们先从总体的角度来看一下 Cinderella 语言具有哪些特性。如下代码所示便是用 Cinderella 语言编写的一个基本函数，该函数主要用来计算两个输入参数的平均值。为了在最大程度上简化编译器的具体开发流程，这里只为该语言设计了最基本的、最简单的几个特性。其中最主要的特性之一就是函数的定义过程。在 Cinderella 语言体系中，需要使用关键字“def”来定义函数，函数的参数列表必须以“(”开始，同时以“)”结束，其内部的形参以“,”进行分隔并依次向后排列。紧接着参数列表的是一个可以被计算的数学表达式，该表达式的实际计算值将会被直接作为函数的返回值。最后，还需要在结尾添加一个“;”以结束该函数的定义过程。

```
average.hs
```

```
def average(x, y) (x + y) * 0.5;
```

目前在 Cinderella 的语言体系中，并没有定义以下特性。

应用程序入口函数

由于没有定义主函数，因此我们只能将基于 Cinderella 语言编写的应用程序源代码编译成静态共享库的形式，以方便在其他应用程序中进行调用。对于 x86 等传统 ISA 指令集来说，静态库中的函数只能通过定义有主函数的应用程序来调用和执行；而对于诸如 WebAssembly 等 V-ISA 指令集来说，静态库中的函数应该如何执行和表现则要根据运行平台上对应虚拟机的具体实现来决定。比如对于 WebAssembly 来说，编译到该 V-ISA 的静态库其内部定义的函数可以直接在浏览器平台上通过 JavaScript 脚本来进行调用和执行，而这与传统指令集架构完全不同。

变量类型

因为不支持显式地为变量设置具体的数据存储类型，所以在函数的实际执行过程中可能会引发一些意想不到的异常。因此，对于类似的弱类型语言来讲，为变量添加适当的默认数据类型转换规则是非常有必要的。但是这里为了简化编译器的实现过程，我们强制所有以字面量形式出现的数据，以及函数参数变量，均对应 C/C++ 语言中的 double 双精度浮点类型。

流程控制语句

为了简化语言编译器的实现过程，这里暂时不为 Cinderella 的语言体系添加基本的流程控制语句，如 if 条件选择语句、for 循环语句等。而这些控制结构则是图灵完备编程语言所必不可少的组成部分。

在了解了 Cinderella 语言的基本特性之后，我们再从整体上来规划开发编译器的主要流程。

由于整个开发流程涉及的步骤过于繁多，因此我们将其分为主要的两大部分来进行讲解。在如图 4-19 所示的第一部分内容中，首先需要构建的是用于对源代码组成结构进行 Token 分析的词法分析器（Lexer），以及用于进行语法分析并生成 AST 结构的语法分析器（Parser），然后通过结合 LLVM 工具链提供的一系列功能函数，我们可以方便地构建用于生成 LLVM-IR 代码的生成器。从总体上看，这一部分功能包含了一个完整编译器前端所具备的所有功能。

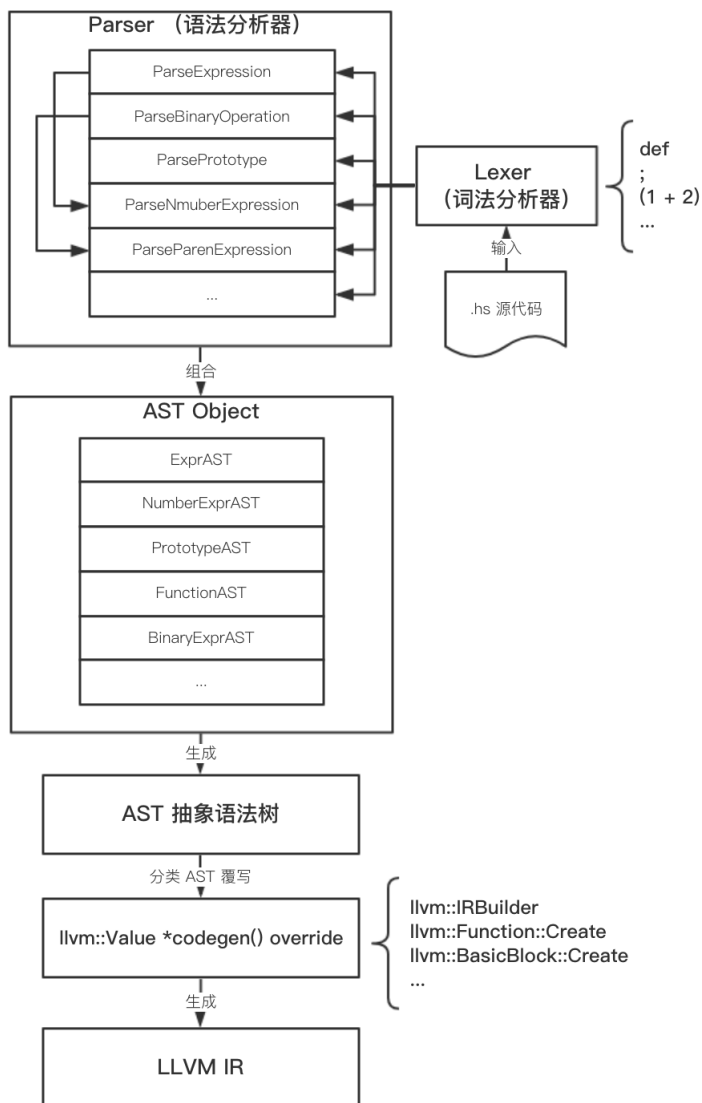


图4-19 Cinderella语言编译器的整体开发流程（第一部分）

在如图 4-20 所示的第二部分内容中，我们将大量接入 LLVM 提供的优化器，以及与各个平台架构相对应的编译器后端。你会发现，完成这部分功能远比开发编译器前端要简单得多，而这一切都得益于 LLVM 工具链已经封装好的各类成熟的优化器组件及后端代码生成器。

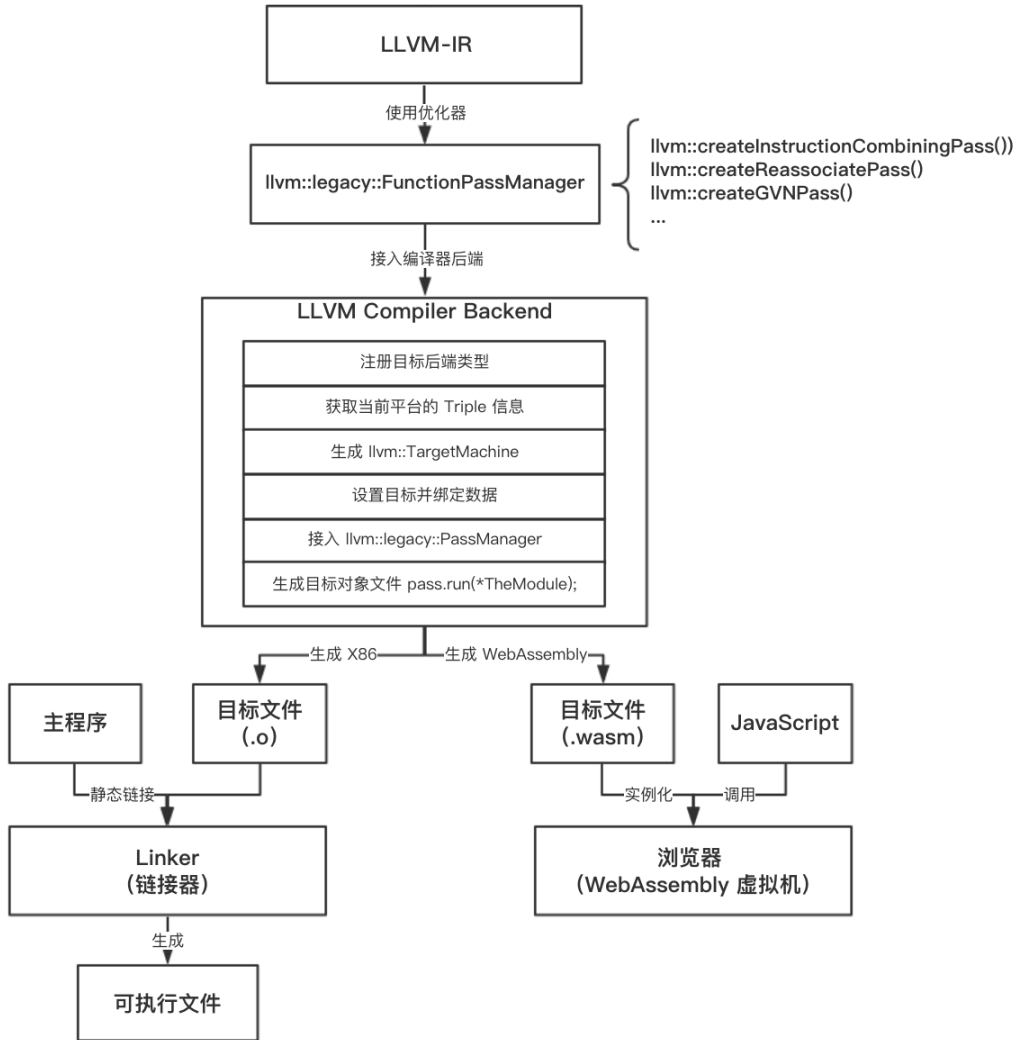


图4-20 Cinderella语言编译器的整体开发流程（第二部分）

在真正着手开发之前，有必要对一些与编程语言相关的概念进行介绍，如果你对这部分内容已经有所了解，则可以跳过 4.2.1 节，直接从 4.2.2 节开始阅读。

4.2.1 图灵完备与 DSL

在平日的技术社区活动或日常工作讨论中，我们经常会听到有人在谈论某种编程语言是否是图灵完备的。那么，究竟什么是图灵完备（Turing-Complete），并且它和编程语言之间又有着怎样的关系呢？用简短的一句话来介绍，就是如果某种事物其本身可以模拟出图灵机所具有的计算模型，那么便可以称该事物是图灵完备的。因此，无论是一台精密的现代计算机、一种编程语言还是一把算盘，只要能够模拟出图灵机所具有的计算模型，那么便可以说它是图灵完备的。而由此引出的另一个问题就是：什么是图灵机，并且它具有怎样的计算模型呢？

图灵机

图灵机是由英国数学家、逻辑学家、计算机科学之父艾伦·图灵在其 1936 年发表的论文《论数字计算在决断难题中的应用》中所提出的一个数学模型。虽然这只是理论上的一个数学模型，但是图灵在论文中却描述了该模型对应的一个假想机器的基本结构，而这个假想机器就是图灵机的原型。图灵实际上是想用机器来模拟人们用纸和笔进行数学计算的过程，他认为这个过程可以被分成以下两种简单的动作。

- 在纸上写下/擦除某种符号。
- 把注意力从纸的某个位置转移到另一个位置。

通过这两种动作，人们便可以完成最简单和最基本的计算过程。图灵机正是为了实现这两种动作而提出的一种假想的机器原型。一个基本的图灵机由以下几个部分组成。

一条无限长的纸带（TAPE）

这条纸带被分为一个个相邻的格子，在每一个格子中都可以写入至多一个符号。纸带的右端可以无限延伸。

一个字符表（SYMBOL）

在这个表结构中存放着所有可以被写入纸带格子中的有限个字符。其中包含一个特殊的空白字符，该字符表示将格子内容置空。

一个读写头（HEAD）

该读写头可以在纸带上左右来回移动。顾名思义，通过读写头可以选择读取或修改纸带上某一个格子中的符号内容，从某种程度上讲，它便是图灵机的 I/O 总线。

一个状态寄存器（REGISTER）

该寄存器主要用于保存图灵机当前所处的状态。图灵机拥有的状态个数是有限的，其中有一个特殊的“停机状态”，该状态表示图灵机当前已经停止运行。

一套控制规则（TABLE）

这套规则用来控制读写头的下一步动作。读写头会根据当前状态寄存器及所在格子的符号内容，从控制规则中找出对应的下一步动作。每一步动作都会按照以下顺序来执行：写入或更新当前格子中的符号内容→向纸带的左/右侧移动或不移动→保持状态寄存器中的当前状态或更新状态。

图灵机的基本结构如图 4-21 所示。在机器开始运行前，在纸带上的格子内部可以事先写入任意的初始化数据来作为图灵机的输入。当计算过程开始时，读写头会根据控制规则按照顺序来修改纸带上各个格子中的符号内容。当控制规则全部执行完毕后，此时纸带上一系列格子组成的序列内容便可以作为输出，直接由人类解码为自然语言。图灵认为，只要上述图灵机模型能够被实现，便可以用来解决任何的可计算问题。那么，什么是可计算问题？我们可以通过两个例子来进行深刻理解。比如，类似于“给出一个正整数，求出它的按位取反结果”的问题便是一个可计算问题，因为我们可以求得该问题的最终结果。而“太阳为什么可以发光”便不是一个可计算问题，因为该问题并不能量化到图灵机的具体计算流程上来。由于可计算问题涉及与可计算理论相关的复杂知识，因此这里不做严格定义。我们只需要知道的是，凡是对于人类可以保证在有限时间内计算出其结果的问题，图灵机同样也可以做到。

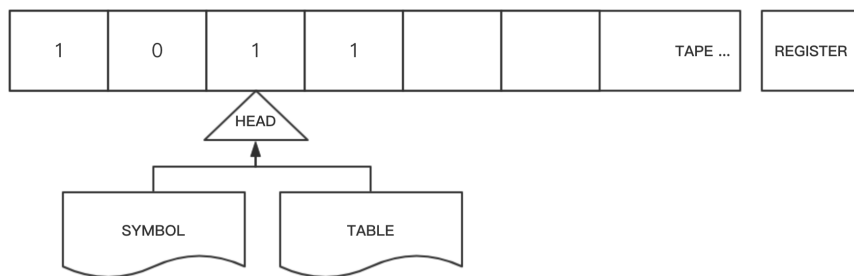


图4-21 图灵机的基本结构

介绍完图灵机的基本概念，我们便可以通过总结图灵机所具有的一些典型特征来给出图灵完备的具体要求。此处我们定义的要求不一定严格，但是却包含了图灵机具有的大部分直观特性，总结如下。

具有流程控制结构

这一特性对应于图灵机中的“控制规则”部分。该特性若体现在编程语言或计算机实现中，则表示对 if/else 等基本流程控制结构的支持与实现。

具有资源处理能力

所谓的资源处理能力，是指能够对从外界输入到系统内部的数据进行计算和处理的能力。这个特性直接对应于图灵机中“读写头”的功能与定位。

具有有限的可操作元素

这里所讲的“有限的可操作元素”，在图灵机中是指可被打印在纸带格子内的符号类型是有限的。该特性若体现在编程语言或计算机实现中，则表示在源代码中和程序运行时能够使用到的基本数据类型是有限的。

具有无限的资源存储能力

在图灵机中，用于记录符号序列的纸带可以无限延伸，而符号序列本身就对应于图灵机所拥有的数据资源。无限长度的纸带若体现在编程语言或计算机实现中，则表示具有无限的资源存储能力。

以上我们总结出了当一种事物想要具备图灵完备特性，即能够模拟出一个图灵机所具有的基本功能时，其需要符合的一些基本条件。但这并不意味着一种事物只有在满足上述所有条件时才能被称为是图灵完备的。因为在这些条件中，有一项在现实世界中是永远无法满足的，那便是“具有无限的资源存储能力”这一特性。为此，从图灵机的广义概念上又延伸出几类可以应用在现实世界中的图灵机类型。其中“线性有界自动机”便成为最常用的一种图灵机类型，该图灵机将其自身的计算过程限制在仅包含输入的那一段纸带上，这使得我们在现实世界中构建图灵机成为可能。而实际上，所有的现代计算机均是该类型图灵机的一种具体实现。

现在我们常用的各类主流编程语言如 C/C++、Python、Java 等均符合图灵完备特性，从本质上讲，它们都具有图灵机最核心的计算能力，只是语言上层的具体表现方式不同而已。为了能够更加直观和深刻地去理解图灵完备的概念，我们可以尝试从一种名为“Brainfuck”的编程语言入手，该语言的整体语法结构和执行原理均与图灵机十分相似。

Brainfuck

如图 4-22 所示为我们使用 Brainfuck 语言编写的一个可以打印出“Hello Word!”字符串的

源代码程序。可以看到，在这段代码中大量重复地出现了诸如“<”“+”等简单字符，这些字符每一个都代表着一种特定的功能指令，并控制应用程序做出相应的动作。从整体上看，这些字符对应于图灵机中“字符表”的概念。在整个 Brainfuck 语言中只定义了 8 种有效的字符指令，这些字符指令可以控制应用程序将数据写入一个一维数组的任意单元内，而用于存放数据的数组结构则直接对应图灵机中的“纸带”。该语言的解释器在运行时会依次读取源代码中的每一个字符，对于有效字符，则会根据其功能来做出相应的动作；而对于无效与空白字符，则会直接将其过滤掉。

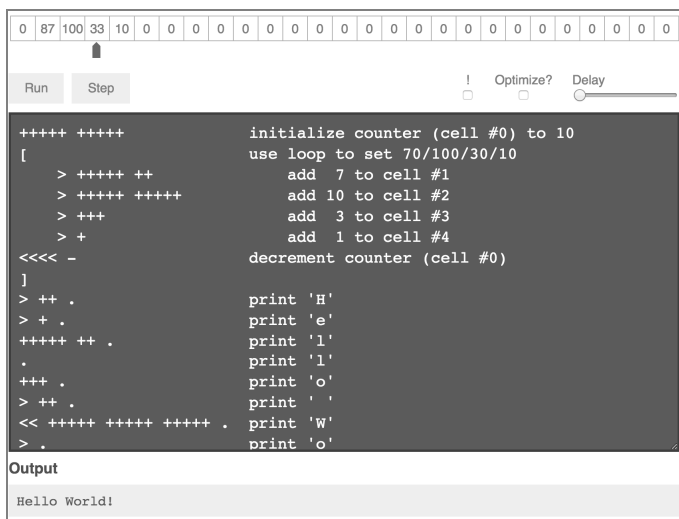


图4-22 Brainfuck程序源代码

在 Brainfuck 中定义的 8 种有效字符指令分别如下。

- “>”: 将当前的数据指针向右侧移动，指向一维数组当前位置的下一个单元。
- “<”: 将当前的数据指针向左侧移动，指向一维数组当前位置的上一个单元。
- “+”: 将当前数据指针指向的数组单元其内部存放的数值加 1。
- “-”: 将当前数据指针指向的数组单元其内部存放的数值减 1。
- “.”: 将当前数据指针指向的一维数组单元其内部存放的数值输出到控制台中。
- “,”: 从外部输入流中接收一个值，并将其存放在数据指针当前指向的一维数组单元内。
- “[”: 判断结构。若数据指针当前所指向的一维数组单元内部存放的值为 0，应用程序

的控制流程则直接跳转至对应的“]”符号后面。

- “]”: 判断结构。若数据指针当前所指向的一维数组单元内部存放的值不为 0，应用程序的控制流程则直接跳转回对应的 “[” 符号后面。

虽然 Brainfuck 是一种极小化的计算机编程语言，但它却是完全匹配图灵完备思想的，因此仅通过该语言我们便可以处理任何可计算任务。整个 Brainfuck 语言体系的组成结构与图灵机十分类似，通过了解该语言的语法特征，我们可以从感性的角度更加直观地体会图灵完备的意义。

从总体上看，基本上常见的高级程序设计语言都是符合图灵完备定义的。但是仍然有一些具有特殊用途的编程语言并不符合该定义，其仅用于特定领域，用来实现特定功能，我们通常将这种语言称为 DSL（Domain-Specific Language，特定领域语言），而将与之相反的编程语言称之为 GPL（General-Purpose Language，通用编程语言）。比如在前端开发中用于构建 Web 网页的 HTML、CSS 语言便属于 DSL。同样的，用于对数据库进行结构化数据查询的 SQL 也是一种 DSL。这些语言并不具有流程控制与数据交互这两种基本的图灵机能力，因此我们无法将其界定为具有图灵完备特性。

但实际上，DSL 与 GPL 的划分界限并不十分清晰。某些语言可能是专门为解决某一领域问题而设计的，但是在技术上却可以被应用到更广泛的领域。相反，同样有一些语言被设计成可以普适性地应用于多个领域，但实际上人们却只愿意将其应用在处理特定问题的场景中。不仅如此，DSL 与图灵完备特性之间也并没有明确的对应关系，比如 PostScript 本身是一种图灵完备的编程语言，但在实践中却经常被狭义地用在“列印图像和文字”这类场景中。因此，从某种角度来说，PostScript 是一种图灵完备的 DSL。

总的来讲，之所以要划分出 DSL 与 GPL 这两种语言类型，是由于在某些特定的问题场景下，虽然 GPL 可以作为解决问题的通用语言，但是却无法以一种更适合的方式来表达解决这些问题的具体方案。如果能够想到一种解决方案可以比现有语言的表达更加精准和明确，那么重新创建一种新的 DSL 而不是重用现有的 GPL 可能是十分有价值的。

4.2.2 简易词法分析器

从本节开始，我们将真正介绍 Cinderella 语言编译器的开发流程。前面介绍过，整个编译器前端是由词法分析器（Lexer）和语法分析器（Parser）两部分组成的。其中词法分析器主要用于将源代码中一系列普通的 ASCII 字符转换为具有特定含义的语言关键字标识，我们首先就从构建它开始入手。

如图 4-23 所示，作为整个编译器链路的第一环，词法分析器会直接作用在输入到编译器内部的源代码文本上。词法分析器首先会以字符为单位将这些代码文本依次读入其内部，然后开始分析并依次找出代码中与特定关键字（比如一个数字类型的表达式、用于流程控制的 `if/else` 关键字，以及用于定义函数类型的 `def` 关键字等）特征相匹配的子字符串结构。对于这些具有特定语义的关键字字符串，我们可以将其称为“Token 字符串”。除此之外，词法分析器在分析过程中还会不断地过滤掉空白字符及换行符等与代码本身不相关的信息，这样便可以保证从其内部输出的每一个字符串都是一个有效的、带有特定语义的 Token 字符串。

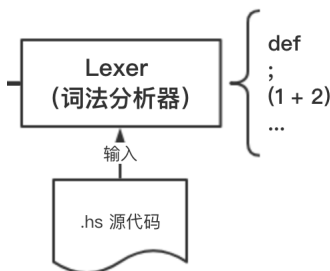


图4-23 词法分析器结构

通常来说，传统词法分析器的构建过程需要经过构建正则表达式、转换至 NFA（不确定的有穷自动机）、转换至最小化 DFA（确定的有穷自动机）、构建词法生成器等一系列复杂的步骤。而为了实现这些步骤，我们还需要了解各类晦涩难懂的自动机的概念。但无论以哪种方式进行开发，词法分析器本身的功能与定位均是不变的。因此，为了能够让读者直观地感受词法分析器在整个编译器链路中的功能与定位，这里将采用最直接和简单的方式来构建用于 Cinderella 语言的词法分析器，仅通过简单的条件判断语句来分析和构建各类 Token 字符串。

```
def average(x, y) (x + y) * 0.5; # 函数定义
```

首先需要定义词法分析器在分析源代码时可能遇到的所有 Token 字符串类型，以便能够进行准确的识别。在上面这行代码中，我们通过 Cinderella 语言定义了一个用于求平均值的函数 `average`，其中使用到了函数定义、基本表达式运算、单行代码注释等 Cinderella 支持的语法特性。参照这行代码，我们可以将词法分析器需要识别的 Token 字符串类型总结为如下几种。

- 用于定义函数的 `def` 关键字：该关键字表于开始一个函数定义过程。它属于 Cinderella 语言在语法特性上的功能关键字。
- 用于表示函数名/变量名的字面量标识符：这里规定标识符只能由阿拉伯数字和区分大小写的英文字母组成，并且只能够以字母作为开头。

- 字面量形式的数字值类型：顾名思义，数字值类型即源代码中直接以字面量形式表示的数字。比如在上面代码中，函数体内使用的小数 0.5 便是一个典型的字面量数字值。为了降低开发难度，这里我们直接使用 C/C++ 中的 `double` 类型变量来存储源代码中出现的字面量数字值。
- 用于分割关键字的空字符：通常，我们都会使用单个空字符（空格）来分割在源代码中出现的各个关键字。但实际上在 `Cinderella` 中，多于一个的连续空字符对于词法分析器来说并没有任何实际作用，其效果等同于单个空字符。因此，当词法分析器在分析源代码时，需要将多余的空字符过滤掉。
- 表示分析结束的 EOF 字符：通常，在 C/C++ 等语言中，EOF（End-Of-File）是一个用于表示“I/O 流已经读取结束”状态的标准宏常量。这里的“I/O 流”可以泛指任何输入/输出流。比如这里对于词法分析器而言，I/O 流便是指从源代码文件中读取源码时编译器打开的文件 I/O 流。当词法分析器从源代码文件中以“字符”为单位逐个读出其内容时，若读到的字符值恒等于 EOF 宏常量，则表明源代码文件中的所有内容均已被读取完毕，词法分析阶段结束。
- 表示单行注释的“#”字符：对于源代码中的所有注释信息，词法分析器都可以通过扫描“#”符号将它们识别出来。这里规定 `Cinderella` 中只支持通过“#”标记的单行注释内容。
- 其他具有特定含义的字符类型：这些字符包括用于在表达式语句中分割参数的“,”（逗号）符号、用于标记函数定义结束的“;”（分号）符号，以及用于组成子表达的位于其左右两侧的“(”和“)”符号等。在实际对源代码的分析过程中，词法分析器会识别出这些符号，并返回这些独立 Token 对应的符号类型。

在分析了词法分析器需要识别的所有符号类型后，接下来我们正式开始编写代码。如下面代码所示，我们创建了一个专门的 `Lexer` 类用来装载词法分析器的所有功能实现，以及其内部的状态变量。可以看到，这里我们将 `Cinderella` 语言中所有可用的 Token 类型全部整合在一个枚举结构中，其中的枚举值可以在整个编译器环境内进行共享。在 `Lexer` 类中，我们声明了一个名为“`_find_token`”的私有方法，用于分析源代码并获取其中的 Token 类型与对应值。该方法在每一次调用时都会从源代码的当前位置开始分析并返回下一个遇到的 Token 枚举类型，该次分析所得的 Token 类型值会被存放到名为“`CurTok`”的类静态变量中。而对于字符串和字面量数字值类型的 Token，由于其可能会含有用户自定义的且不属于 `Cinderella` 语义的数据元素，因此这里将它们的具体值分别存放在名为“`IdentifierStr`”和“`NumVal`”的静态变量中。需要注

意的是，为了便于内存管理，这里会将包括 `Lexer` 类在内的接下来创建的所有其他类成员均标识为静态类型。

```
// 词法分析器需要识别的所有 Token 枚举类型
enum Token {
    tok_eof,
    tok_def,
    tok_identifier,
    tok_number,
    tok_separator,
    tok_left_paren,
    tok_right_paren,
    tok_expr_end
};

// 创建 Lexer 类
class Lexer {
    Lexer() = default;
    ~Lexer();

private:
    // 用于分析和识别 Token 类型的核心私有方法
    static int _find_token();

public:
    // 存放当前识别出的字符串关键字或标识符
    static std::string IdentifierStr;
    // 存放当前识别出的数字字面量值
    static double NumVal;
    // 存放当前识别出的 Token 其对应的枚举类型
    static char CurTok;

    // 对外暴露给语法分析器的用于获取 Token 的方法
    static int GetNextToken() {
        return CurTok = _find_token();
    };
};
```

接下来，我们把目光放到“`_find_token`”这个核心私有方法的实现细节上，该方法的具体实现代码如下。

```
int Lexer::_find_token() {
    static int LastChar = ' ';

    // 读入新的字符并过滤掉空白字符（空格）
    while(isspace(LastChar)) {
        // 该方法每次调用返回源码中的一个字符
        LastChar = IOInterface::ReadCharacterSource();
    }

    // 判断当前读入的字符是否为英文字符
    if (isalpha(LastChar)) {
        Lexer::IdentifierStr = LastChar;
        // 连续读入后续的英文/数字字符，并拼接成字符串形式的标识符
        while (isalnum((LastChar = IOInterface::ReadCharacterSource()))) {
            Lexer::IdentifierStr += LastChar;
        }

        // 判断标识符是否为特定类型的关键字
        if (Lexer::IdentifierStr == "def") {
            return tok_def;
        }

        if (Lexer::IdentifierStr == "extern")
            return tok_extern;
        // 若不是则返回通用类型，即标识符类型
        return tok_identifier;
    }

    // 判断当前读入的字符是否为数字字符
    if (isdigit(LastChar)) {
        std::string NumStr;
        bool validNumStr = true;
        do {
            // 检查当前拼接成的字面量数字值是否合法（小数点是否过多）
            if (std::count(NumStr.begin(), NumStr.end(), '.') > 1)
                validNumStr = false;

            NumStr += LastChar;
            // 更新输入的字符
            LastChar = IOInterface::ReadCharacterSource();
        } while (isdigit(LastChar) || LastChar == '.');
```



```

    } while (isdigit>LastChar) || LastChar == '.');

    if (validNumStr) {
        // 将字面量数值转换为双精度浮点数
        NumVal = std::strtod(NumStr.c_str(), 0);
        return tok_number;
    }
}

// 判断当前读入的字符是否为注释
if (LastChar == '#') {
    // 过滤掉注释部分，直至遇到换行符
    do LastChar = IOInterface::ReadCharacterSource();
    while (LastChar != EOF && LastChar != '\n' && LastChar != '\r');

    if (LastChar != EOF)
        // 过滤掉注释后继续寻找下一个Token
        return _find_token();
}

// 判断当前读入的字符是否为左/右小括号、逗号和“分号”这四种类型的符号
std::vector<char> vec{'(', ')', ',', ';'};
if (std::find(vec.begin(), vec.end(), LastChar) != vec.end()) {
    Token t;
    // 返回对应的枚举类型
    switch (LastChar) {
        case '(':
            t = tok_left_paren;
            break;
        case ')':
            t = tok_right_paren;
            break;
        case ',':
            t = tok_separator;
            break;
        case ';':
            t = tok_expr_end;
            break;
    }
    LastChar = IOInterface::ReadCharacterSource();
}

```

```
        return t;
    }

    // 判断当前读入的字符是否为 EOF，即表示已读取完毕
    if (LastChar == EOF)
        return tok_eof;

    // 若以上条件均不符合，则直接输出该字符对应的 ASCII 码值
    int ThisChar = LastChar;
    LastChar = IOInterface::ReadCharacterSource();
    return ThisChar;
}
```

当外部调用者每次调用 `GetNextToken` 方法时，词法分析器都会通过我们定义在另外一个类中的 `IOInterface::ReadCharacterSource` 方法从源代码文件 I/O 流的当前位置读取一个字符。然后通过判断该字符的具体内容，词法分析器会通过不同的处理策略来组合并筛选出不同的 `Token` 枚举类型。比如，如果当前第一个被读入的字符是一个字面量数字类型值，那么词法分析器会假设接下来的 `Token` 类型为数值值并继续从源代码中读入任意数量的字符，直至遇到用于分割 `Token` 的空白字符。所有这些被读入的字符将会作为一个整体，在词法分析器内部进行相应的规则检验（比如验证是否含有超过 2 个的小数点符号）并被拼接转换为一个双精度类型的浮点数（如果通过检验）。可以看到，源代码中也对后续依次读入的字符进行了适当的类型检测，通过判断它们是否具备组成一个浮点数所必需的特征，比如“字符内容必须为字面量数字值或小数点类型”等特征，我们可以更高效地完成对源代码中 `Token` 的分类和筛选过程。

除需要对符合特定枚举类型特征的 `Token` 进行分类外，其他没有被成功归类的字符将直接返回其对应的 ASCII 符号。而对于诸如“+”“-”“*”“/”等各类用于数学表达式运算的运算符 `Token`，将会放在语法分析器中进行处理。

至此，这个简易的词法分析器便开发完成了。总的来说，词法分析器的主要功能与定位是对源代码中各类具有明确语义的 `Token` 字符串进行归类与提取，并同时剔除其中对应用程序运行没有任何影响的字符部分，比如多余的空白字符、注释（可选）等。词法分析器本身并不关心源代码中关键字的实际使用语法是否正确，而将其成功归类后，却可以最大程度地方便后续语法分析器对源代码的语法分析、生成 AST 数据等多个步骤的进行。

在正式开发语法分析器之前，我们先来了解两种语法分析器经常使用的数据处理算法。通过这些算法，我们能够以更加成熟和通用的方式来解决语法分析器开发过程中遇到的一些问题。

4.2.3 RDP 与 OPP 算法

本节将会介绍 RDP 和 OPP 这两种语法分析器常用算法。其中，RDP 算法主要用于对源代码中各类 Token 元素的语法组成结构进行分析和校验，并生成对应的 AST 数据结构；而 OPP 算法则主要用于处理数学表达式中运算符的计算优先级问题。

RDP 算法

借助 RDP（Recursive Descent Parser，递归下降解析器）算法，我们可以让语法分析器按照自顶向下的方式来分析源代码中各类 Token 元素的语法组成结构。通常来说，对于一个语法分析器而言，它可以采用两种常用的方式来对源代码进行语法分析。第一种是自顶向下的分析方式，该方式将会以“非终结符”作为起始分析入口，并不断对该非终结符进行分解，直至找到最终能够匹配上的终结符展开式形式；第二种与第一种正好相反，采用的是自底向上的分析方式，该方式会以一个个“终结符”Token 作为起始分析入口，然后通过不断地将终结符进行合并，直至最后匹配上目标非终结符形式。

对于终结符和非终结符的相关概念，在本书第 1 章的内容中有所提及，下面我们再来回顾一下。比如这里使用符号“E”来表示一个基本的数学表达式类型，同时规定在该表达式中仅允许使用“+”这一种数学表达式运算符。那么，我们可以写出这个数学表达式的多种组成形式，如下所示。

```
E -> 1
E -> 1 + 2
E -> 1 + 2 + 3
...
```

其中，第一行的数学表达式（E）仅由一个单独的字面量数字值组成；第二行和第三行的数学表达式分别在上一行表达式的基础上通过“+”运算符拼接了新的子表达式，即单一的字面量数字值。接下来，我们将上述表达式中出现的字面量数字值（Token）使用符号“d”进行替换，并同时将该表达式“E”的组成规律以递归展开的形式表达出来。经过整理后，表达式“E”的符号组成形式如下所示。在该符号表达式中，符号“E”为一个非终结符；符号“d”为对应的终结符；符号“|”表示“逻辑或”运算，用于组合出表达式“E”的所有可能组成形式。

```
E -> d | E + d
```

综上所述，这个符号表达式表明：符号“E”对应的具体表达式可以由单独的终结符“d”（单一数字值）组成，或者由非终结符“E”（另一个子表达式）与终结符“d”经过“+”运算符拼接而成的表达式组成。

从词法分析器的角度来看，所谓的“终结符”实际上是指可以与某种 **Token** 类型直接对应的符号类型；而“非终结符”则一般是指语言定义上的某种语法结构，比如表达式、函数这类常见的语法组成部分。我们一般会通过递归的形式来描述非终结符的具体组成结构，因此，比如对于一个简单的表达式类型非终结符，其对应的终结符展开式则会有多种类型。当语法分析器在处理这些复合型的语法结构时，需要找到一种非终结符的具体展开形式，能够使得当前源代码中基于各类 **Token** 组成的语法结构与该展开形式相匹配。

通常来说，我们会使用一种名为“BNF”的语言范式来描述编程语言特定语法特性的符号组成结构。比如对于上述非终结符“E”对应的数学表达式，我们可以使用 BNF 范式将其组成结构改写成如下形式。

```
<E> ::= d | <E> + d
```

可以看到，实际上基于 BNF 的写法，与我们之前使用的自定义写法没有任何本质上的差别，但是从总体上来看，它更加规范。在 BNF 的语法中，我们需要将非终结符全部放置在“::=”符号的左侧，并通过一对尖括号“<>”将其包裹在其中。在“::=”符号的右侧放置了用于描述该非终结符具体组成规则的符号表达式，在这里可以使用任意已经定义的非终结符、终结符，以及用于描述组合选择逻辑的“|”符号，甚至自定义字符。比如，一个用于描述仅含有数学加减运算（仅含有“+”和“-”这两种运算符）的表达式其语法结构的 BNF 可以表示为如下形式。

```
<expr> ::= <expr> + <factor>
          | <expr> - <factor>
          | <factor>
<factor> ::= ( <expr> )
          | Number
```

可以看到，这里定义了两种非终结符。从下往上看，第一种名为“factor”的非终结符表示可以被放置在该类型表达式中的最小原子类型。所谓最小原子类型，可以将其理解为：若将该类型的数据放置在整个表达式中的任何位置上，都不会影响其自身的实际计算结果值。在这里最小原子类型可以是单个的数字值类型（**Number**），也可以是被放置在括号内（决定了求值的优先级）的一个子表达式（<expr>）。第二种名为“expr”的非终结符作为整个表达式的顶层终结符，描述了含有“+”和“-”运算符的子表达式类型。接下来，我们将通过一个满足上述 BNF 范式的具体表达式例子来介绍 RDP 算法的详细执行流程。

```
9 - (2 + 5)
```

由于 RDP 算法是一种自顶向下的语法分析方法，因此我们需要从整个 BNF 范式顶层的非终结符开始进行匹配。整个算法的思路是，将表达式中的每一个独立 **Token** 分别与整个 BNF 范

式组成中的每一个子项依次进行比较，直至找到正确、合理的展开规则。在比较的过程中，需要将遇到的非终结符递归地展开成可能的终结符组成形式，然后将 Token 与这些终结符组成形式依次进行比较。其实对于上述 BNF 范式，由于其在非终结符的可能组成中直接使用了该非终结符本身，因此在使用 RDP 算法进行语法解析时会导致出现“左递归”的问题，即语法分析器在分解非终结符并进行 Token 匹配时会陷入一个死循环。为了解决该问题，首先需要通过如图 4-24 所示的文法变换公式，移除该 BNF 范式中会产生左递归的部分文法。

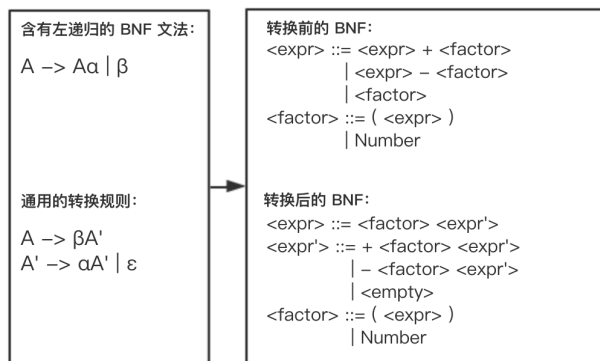


图4-24 用于消除左递归的BNF文法变换公式

由于涉及过多的理论性知识，因此关于移除左递归文法的具体原理和流程，这里我们不做过多介绍，有兴趣的读者可以自行查阅相关资料做进一步了解。当通过上述转换规则将 BNF 中的左递归文法消除后，便得到了如下所示的 BNF 范式。此时，我们便可以通过 RDP 算法对之前的数学表达式进行语法解析了。

```
<expr> ::= <factor> <expr'>
<expr'> ::= + <factor> <expr'>
           | - <factor> <expr'>
           | <empty>
<factor> ::= ( <expr> )
           | Number
```

首先需要将上述数学表达式拆分成若干个不同类型的 Token。这一步骤实际上可以由词法分析器来帮助完成。拆分后的 Token 如图 4-25 所示，其中的字面量数字值对应于 BNF 范式中的终结符类型“Number”，其他符号则直接与 BNF 范式中的相同符号一一对应。

9	-	(	2	+	5	)

图4-25 上述表达式对应的Token序列

基于 RDP 算法，语法分析器会从处于 BNF 范式顶层的非终结符“`<expr>`”开始尝试依次匹配这些 Token。如图 4-26 所示，非终结符“`<expr>`”是由另外两个名为“`<factor>`”和“`<expr'>`”的非终结符组成的，因此按照顺序，需要首先匹配第一个 Token 字符，即判断字面量数值值“9”是否可以与这两个组成“`<expr>`”的非终结符中的任意一个相匹配。而在实际匹配时，还需要将对应的非终结符展开成可能的终结符组成形式。比如对于非终结符“`<factor>`”来说，在实际进行匹配时，需要将数字“9”分别与该非终结符的两种可能组成形式进行比较。由于数字“9”无法与第一种组成形式“`(<expr>)`”正确匹配，因此会继续与其第二种组成形式即“`Number`”类型的终结符进行匹配，这里则正好可以匹配上。至此，我们得到第一个 Token 对应的语法匹配展开式如图 4-26 所示。

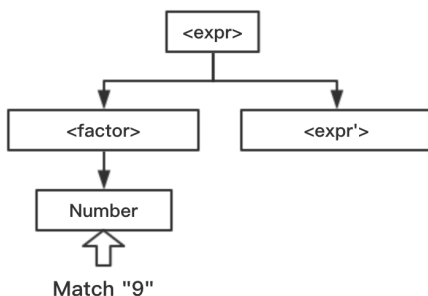


图4-26 第一个Token的语法匹配展开式

接下来继续尝试匹配第二个 Token 字符，即一个减法运算符“-”。这里由于非终结符“`<factor>`”的可能组合形式在之前的过程中已经被全部比较完毕，因此按照 RDP 算法的思路，我们需要将当前“比对指针（Descent Ptr）”的位置回溯到上一层级，并开始对右侧的非终结符“`<expr'>`”进行类似的匹配过程。如图 4-27 所示，我们将该 Token 对应的“-”符号与非终结符“`<expr'>`”的三种可能组成形式依次进行比较，最后可以正确匹配的是其第二种组成形式。

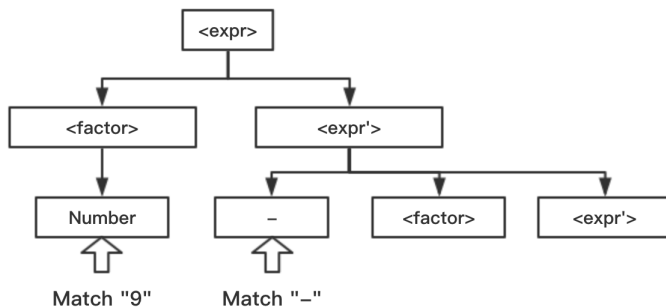


图4-27 加入第二个Token后的语法匹配展开式

按照同样的方式，继续匹配以括号形式组成的子表达式部分。如图 4-28 所示，在非终结符“factor”对应的子表达式组成形式中，我们匹配上了括号子表达式起始的“左括号”部分。

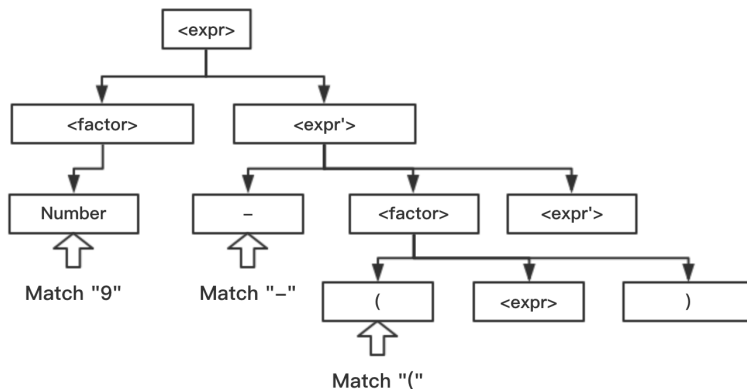


图4-28 加入第三个Token后的语法匹配展开式

对于一个数学表达式的语法组成结构来讲，在一对小括号内可以放置一个任意类型的数学表达式，甚至在该表达式中也可以再放置由小括号包裹起来的子表达式。如图 4-29 所示，在下一步匹配小括号子表达式的过程中，我们便以递归的方式再一次展开该 BNF 范式的顶层终结符“expr”，并在其可能的组成形式中寻找字面量数字值“2”的正确匹配位置。

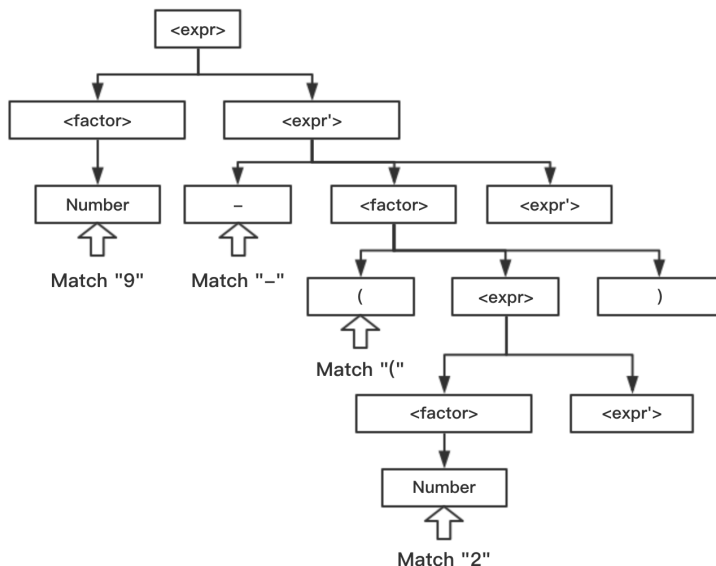


图4-29 加入第四个Token后的语法匹配展开式

接下来，继续匹配括号子表达式中的“+”运算符，如图 4-30 所示。

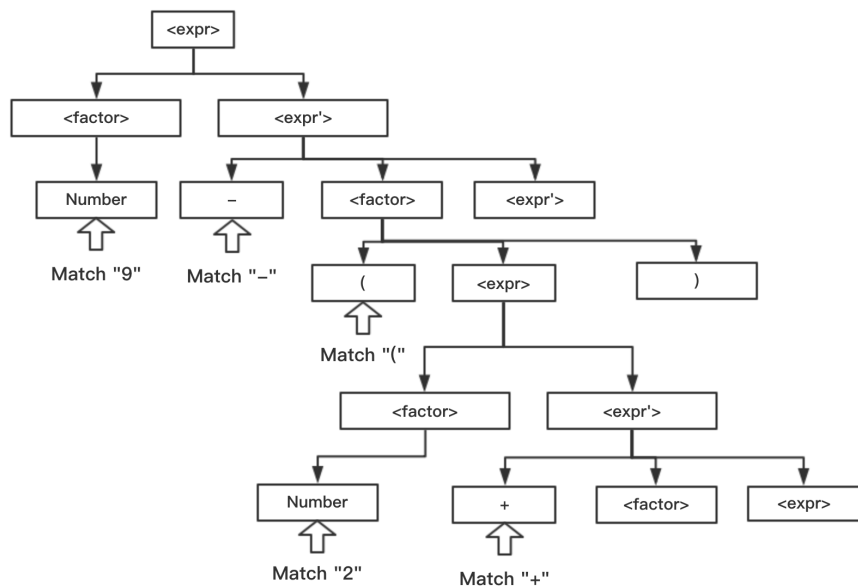


图4-30 加入第五个Token后的语法匹配展开式

继续匹配括号子表达式中值为“5”的字面量数字值，如图 4-31 所示。

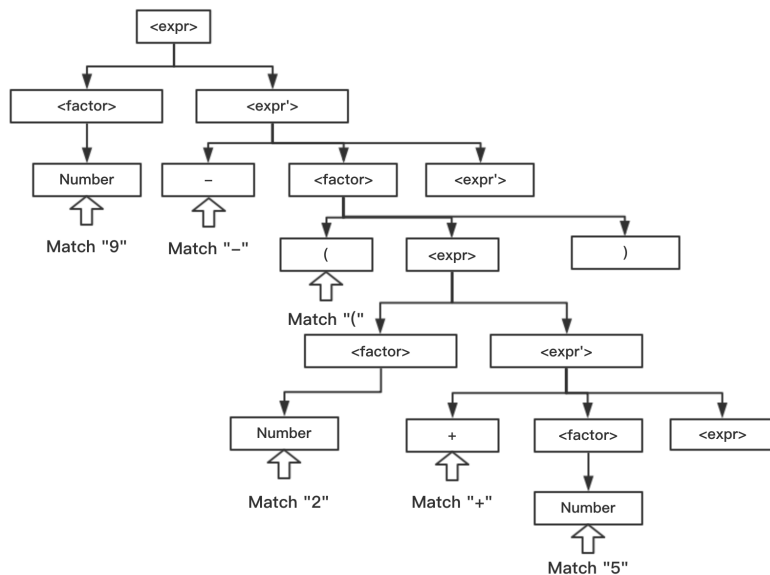


图4-31 加入第六个Token后的语法匹配展开式

Symbol	Precedence
+	1
*	2

图4-33 运算符优先级示例

整个 OPP 算法的核心思路用通俗的话来讲，就是“多向后查看一个运算符”。即在对当前子表达式进行求值前，需要先向后查看下一个子表达式中运算符的优先级。然后通过比较该运算符优先级与当前子表达式运算符优先级的大小关系，来判断是直接对当前子表达式进行求值操作，还是对后面的具有更高优先级运算符的子表达式进行求值操作。整个“向后看”的过程会以递归的方式不断进行，并且只有当判断出某个子表达式的运算符优先级大于或等于其左、右两侧子表达式中的运算符优先级时，算法才会真正地进行子表达式的求值过程。

这里假设有一个 `parse_expression_precedence` 函数，在该函数中实现了 OPP 算法。函数共接收两个参数，其中第一个参数用于存放当前正在分析运算符的下一个运算符的左操作数；第二个参数用于存放当前需要进行判断的最小运算符优先级。如图 4-34 所示为通过该函数对之前给出的数学表达式进行运算符优先级匹配的具体过程。

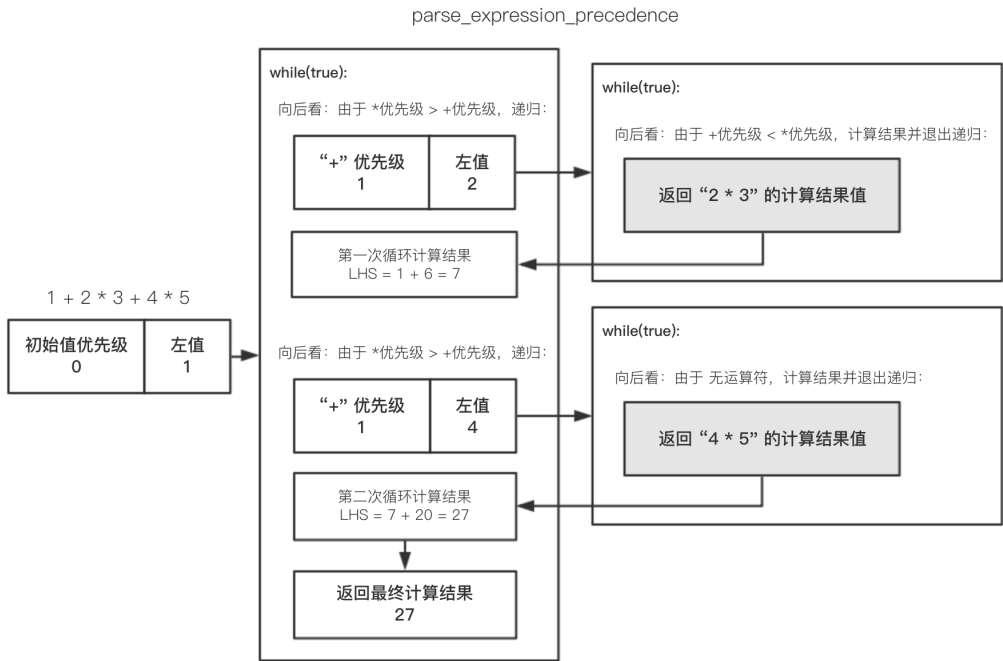


图4-34 上述数学表达式基于OPP算法进行的运算符优先级匹配过程

可以看到，图 4-34 中每一个同时具有“优先级”和“左值”两个字段的方格均表示一次对 `parse_expression_precedence` 函数的调用过程；而带有“`while(true):`”循环结构的大方格则表示该函数内部的具体执行流程。这里我们在 `parse_expression_precedence` 函数的初次调用过程中又分别进行了两次该函数的递归调用，每一次递归的发生条件都是由于在表达式的分析过程中，又遇到了比当前子表达式运算符优先级更高的后置运算符。按照 OPP 算法的执行流程，我们需要优先对整个表达式中具有最高优先级运算符的子表达式进行求值。

参考上述算法的具体执行流程，我们可以将该函数的实现细节用如下伪代码进一步表示出来。

```
parse_expression_precedence (MinPrecedence, LHS):
    while (true):
        TokPrec := 取当前运算符的优先级()

        // 假设无运算符，则 TokPrec 返回 -1
        if (TokPrec < MinPrecedence):
            return LHS

        BinOp := 取当前运算符()
        RHS := 获取该运算符的右值()
        NextPrec := 获取下一个运算符()

        if (TokPrec < NextPrec):
            RHS := parse_expression_precedence(TokPrec, RHS)

        LHS := 计算表达式的值(LHS, RHS, BinOp)
```

4.2.4 AST（抽象语法树）

在介绍完构建语法分析器需要使用的两种算法后，我们再来聊一聊当语法分析器完成对源代码的语法与语义分析后，其生成的最终产物，即一种名为“AST”的数据结构。

AST（Abstract Syntax Tree，抽象语法树）作为语法分析器的语法分析结果产物，其主要目的是希望能够通过一种结构化的树形数据结构来描述源代码的具体语法组成。与 LLVM-IR 一样，AST 作为一种源代码的语法表示方式，它也可以存在于多种具体的表现形式中。比如，可以将 AST 中的各类标签和结构名，以 ASCII 字符的形式直接打印成能够被人类识别的可读文本形式。而在接下来为 Cinderella 构建语法分析器的过程中，对应的 AST 数据结构将会以 C++ 对象的形式被存放在内存中。

对应于源代码的 AST 会捕捉和记录代码的实际执行流程，这在一定程度上有利于后期编译器后端生成机器码的过程。实际上，在实现语法分析器的过程中，我们希望编程语言中的每一

种语法构造类型（Token）都能够拥有一个与之相对应的 AST 构造对象，并且该构造对象需要严格地模拟该语言中对应语法特性的具体组成结构。比如对于一个常见的仅含有四则运算符的数学表达式来说，我们希望对应的 AST 构造对象能够由三个部分组成：一是用于存放具体运算符实体的字符变量“Op”；二是用于存放运算符左操作数的变量“LHS”；三是用于存放运算符右操作数的变量“RHS”。其中用于表示操作数的两个变量均为复合类型。对于编程语言中的其他语法特性，都可以通过类似的方式来创建其所对应的 AST 结构。

4.2.5 简易语法分析器

现在我们步入正题，从零开始构建如图 4-35 所示的用于 Cinderella 语言的语法分析器。在这一部分内容中，我们将编写多个用于分析不同独立语法单元的功能函数，这些函数彼此之间会根据 BNF 范式的相关规则相互递归调用。对于整个语法分析器部分，需要编写的内容可以分为两类：第一类是对应于各语法构造类型的 AST 对象定义；第二类是用于对源代码进行语法分析，并生成对应 AST 结构的各种语法分析方法。

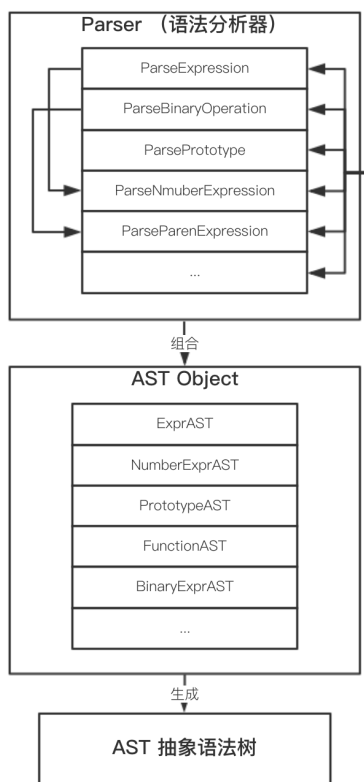


图4-35 Cinderella语言编译器链路中的语法分析器部分

为了能够从整体上了解语法分析器支持的语法特性，首先给出如下一组 BNF 范式，这组范式从语法的元素组成结构上描述了该语言当前支持的各类语法特性。

```
# 数字值表达式
<numberexpr> ::= Number

# 标识符表达式
<identifierexpr> ::= Identifier | Identifier '(' (<expression>|(<expression> ',' )* ')'

# 括号表达式
<parenexpr> ::= '(' <expression> ')'

# 主要表达式类型
<primary> ::= <identifierexpr> | <numberexpr> | <parenexpr>

# 数学运算的右子式
<binoprhs> ::= ('+'|'-'|'*'|'/' <primary>)*

# 表达式
<expression> ::= <primary> <binoprhs>

# 函数原型
<prototype> ::= Identifier '(' (Identifier | Identifier ',' )* ')'

# 函数定义
<definition> ::= 'def' <prototype> <expression>

# 外部引用
<external> ::= 'extern' <prototype>

# 用于命令行的顶层表达式
<toplevelexpr> ::= <expression>

# 用于源代码的顶层表达式
<top> ::= <definition> | <external> | <expression> | ';'

```

可以看到，为了实现用于该 MVP 版本 Cinderella 语言的静态编译器，我们需要在语法分析器部分完成的工作还是非常多的。

总的来看，除对应非终结符部分的各种基本语法元素（字面量数字值、字符串值等）定义外，在该版本的实现中还将以支持函数和数学表达式这两部分语法特性为主。上面所示的 BNF 范式描述了在该语言 MVP 规范中定义的各类可用语法特性其可能的终结符组成结构。如下给出了对应于每个范式的一种可能的实际源码形式。

```
# 数字值表达式: 10
```

```
# 标识符表达式: variableA / funcA()
# 括号表达式: (1 + 2 + 3)
# 主要表达式类型: variableA / 10 / (1 + 2) * 3
# 数学运算的右子式: + 1 * 3
# 数学表达式: (1 + 2 + 3) + 1 * 3
# 函数原型: funcA (x, y)
# 函数定义: def funcA (x, y) x + y
# 外部引用: extern funcB
```

构建 AST 对象子类

这里我们只选择性地介绍其中几种语法特性的具体实现过程，关于其他语法特性的实现细节可以参考 [Github](#) 仓库中的完整项目代码。首先需要创建一个可以用于所有 AST 对象的基本父类，然后在该类的基础上创建需要实现的 AST 对象子类。父类部分的代码如下。

```
class ExprAST {
public:
    virtual ~ExprAST() = default;
};
```

可以看到，在该类中暂时并没有加入任何成员属性和方法。但是让后续创建的所有 AST 对象类都继承自该父类，可以使后面的代码编写过程变得更加简单。不仅如此，诸如“错误处理”等可以通用于所有 AST 对象类的公共方法，也可以被放置在该父类中以供其所有派生类直接使用。从理论上讲，对于实现语法分析器这类功能极其复杂的应用程序来说，应该在其实现代码中大量使用诸如“工厂模式”等优秀的设计模式，以便更好地组织代码。而这里为了简化实现流程，同时希望能够将语法分析器的具体实现原理更加直观地展现出来，所以我们采用了最直接的由上至下的方式来编写和组织代码。

当父类编写完成后，我们便可以开始着手构建各个 AST 对象的子类结构了。这里将以实现语法分析器的数学表达式语法分析功能为例，来介绍包括链接优化器、生成目标代码等在内的后续一系列基本的编译器流程。顾名思义，所谓的数学表达式必定是由一系列的基本数字表达式加上用于指定具体计算类型的数学运算符组成的。因此，首先需要将上述 BNF 范式中非终结符“`numberexpr`”对应的数字表达式类型的 AST 对象构建出来。如下面的代码所示，我们基于 ExprAST 父类构建了一个名为“NumberExprAST”的全新子类。

```
class NumberExprAST : public ExprAST {
    // 用于存放对应的数字值
    double Val;
```

```
public:
    NumberExprAST(double val) : Val(val) {}
};
```

在该类的定义中，仅有的一个成员属性“Val”用于存放对应该类型 AST 节点对象所持有的数字值。需要注意的是，为了方便起见，我们直接使用双精度浮点变量来存储源代码中出现的所有字面量数字值。按照同样的思路，继续构建用于表示变量表达式和数学表达式 AST 对象的对应类结构。首先是用于表示变量表达式的 VariableExprAST 类结构。这里需要注意和区分的是，变量其本身便是一种标识符。

```
class VariableExprAST: public ExprAST {
    // 用于存放对应的变量标识符
    std::string Name;

public:
    VariableExprAST(const std::string &name) : Name(name) {}
};
```

这里我们在定义 VariableExprAST 类时为其添加了一个名为“Name”的字符串变量，该变量将用于存放对应该类型 AST 节点对象的变量标识符，即变量名。接下来继续构建用于表示数学表达式的 BinaryExprAST 类结构。

```
class BinaryExprAST: public ExprAST {
    char Op;
    std::unique_ptr<ExprAST> LHS, RHS;

public:
    BinaryExprAST(char op, std::unique_ptr<ExprAST> LHS, std::unique_ptr<ExprAST> RHS)
        : Op(op), LHS(std::move(LHS)), RHS(std::move(RHS)) {}
};
```

按照之前所讲的，对于数学表达式类型的 AST 对象，我们将通过三个不同的成员属性来表示。其中，字符类型的“Op”属性用于存放该表达式的数学运算符；基于 ExprAST 类型的“LHS”和“RHS”属性则主要用于存放该表达式的左子式和右子式。由于上面创建的所有 AST 对象类均继承自 ExprAST 类，因此这里可以通过树形展开的方式来描述一个含有多运算符的数学表达式。比如对于表达式 $1 + 2 * 3 + 4 * 5$ ，便可以通过如图 4-36 所示的 BinaryExprAST 类对象关系，来从数学表达式的语法组成上描述该表达式的具体组成结构。

可以看到，这里将数学表达式以 AST 的形式展现出来，其中直接用到了 NumberExprAST 和 BinaryExprAST 两个 AST 节点对象。作为对应终结符的 AST 节点对象，NumberExprAST 主

要用于存储整个数学表达式中的所有字面量数字值；而 **BinaryExprAST** 则从数学表达式的语法结构上将这些数字值和运算符结合在一起。需要注意的是，AST 本身只负责展示各语言结构的语法组成关系，对诸如数学表达式的运算顺序等语义上的属性并不会被包含其中。比如对于上述 AST 结构，在实际进行表达式求值时，我们会从该树最底层的终结符开始进行计算，因此会首先计算子表达式 $2 * 3$ 的值，然后计算子表达式 $4 * 5$ 的值，依此类推。AST 的树形结构直接决定了各语法单元的执行顺序，而类似表达式执行顺序这种语义上的规定，则需要由语法分析器在进行语法分析时，将其直接整合到生成的 AST 中。

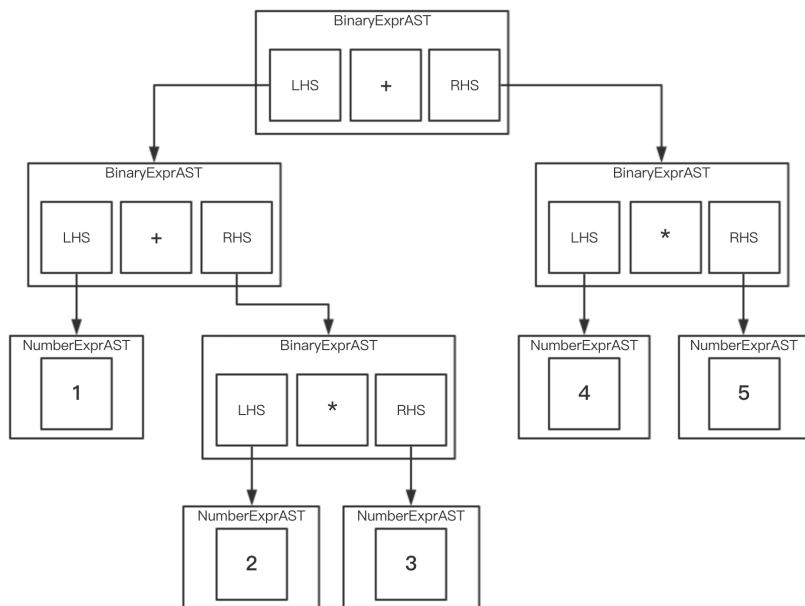


图4-36 上述数学表达式对应的AST结构

构建 Parser 语法分析器

至此，我们已经编写好了所有需要用到的 AST 节点类，接下来将开始构建用于进行源代码语法分析的核心语法分析器。首先构建一个用于承载所有语法处理函数的 **Parser** 类。为了便于内存管理，这里将直接以静态成员的方式来定义这些语法处理函数。

```

class Parser {
public:
    Parser () = default;
    ~Parser () = default;
    // 所有的语法处理函数将会被放置于此
}
  
```


接下来，我们将编写终结符对应的语法分析函数。如下所示的第一个函数会将从源代码中读入的字面量数字值，直接包装成 `NumberExprAST` 类对应的 AST 对象节点类型。这里没有特别复杂的逻辑，但需要注意的是，语法分析器在每次“封装”完一个新的 AST 节点对象后，都需要手动调用 `Lexer::GetNextToken()` 方法来更新在词法分析器中缓存的源代码对应当前位置的字符（获取下一个 `Token` 并将其缓存）。

```
static std::unique_ptr<ExprAST> ParseNumberExpr() {
    auto Result = llvm::make_unique<NumberExprAST>(Lexer::NumVal);
    // 更新下一个缓存字符
    Lexer::GetNextToken();
    return std::move(Result);
}
```

当定义好用于生成终结符对应 AST 节点对象的方法后，下面我们继续向上层封装。如下代码定义的 `ParsePrimary` 方法，将会根据词法分析器中当前缓存的源代码字符（`Lexer::CurTok`）类型来确定需要调用的语法分析方法。比如，当词法分析器告知 `ParsePrimary` 方法当前其缓存的是一个字面量数字值类型的终结符时，该方法将通过分支选择语句调用 `ParseNumberExpr` 方法，并将该字符直接包装成对应类型的 AST 节点对象。而对于“左小括号”这类可能会构成数学表达式（`<binoprhs>`）非终结符的 `Token` 字符，该方法则会通过调用 `ParseParenExpr` 方法来对其进行基于 RDP 算法实现的语法分析过程。

```
static std::unique_ptr<ExprAST> ParsePrimary() {
    switch (Lexer::CurTok) {
        case tok_number:
            return ParseNumberExpr();
        case tok_left_paren:
            return ParseParenExpr();
        default:
            return nullptr;
    }
}
```

如下代码所示为 `ParseParenExpr` 方法的具体实现过程。该方法内部的语法分析流程直接参考了我们之前给出的括号表达式 BNF 范式“`<parenexpr> ::= '(' <expression> ')'`”。可以看到，在该方法内部又调用了用于解析另一个表达式类型非终结符语法结构的 `ParseExpression` 函数，而这正是由于所有被放置在小括号中的内容都可以作为一个独立的表达式类型来进行语法解析。

```
static std::unique_ptr<ExprAST> ParseParenExpr() {
    // 让词法分析器获取源代码中的下一个字符（进入小括号内部）
```

```

    Lexer::GetNextToken();
    // 递归下降
    auto V = ParseExpression();
    if (!V)
        return nullptr;

    if (Lexer::CurTok != tok_right_paren) {
        return ExprAST::LogError("expected ')'");
    }
    Lexer::GetNextToken();
    return V;
}

```

下面给出的是上述成员函数 `ParseExpression` 的实现细节。根据表达式类型对应的 BNF 范式 “`<expression> ::= <primary> <binoprhs>`”，可以知道，一个表达式是由 `<primary>` 类型代表的左子式和 `<binoprhs>` 类型代表的右子式组成的。当然，最简单的左子式可以只是一个字面量数值值。

```

static std::unique_ptr<ExprAST> ParseExpression() {
    // 获取表达式的左子式
    auto LHS = ParsePrimary();
    if (!LHS) {
        return nullptr;
    }
    // 根据 OPP 算法递归解析该表达式
    return ParseBinOpRHS(0, std::move(LHS));
}

```

在接下来的 `ParseBinOpRHS` 方法中，我们便开始通过 OPP 算法根据含有的数学运算符优先级递归地解析数学表达式，并根据 `ParseExpression` 方法提供的表达式初始左值来生成整个表达式对应的 AST 结构（见图 4-36）。此处 `ParseBinOpRHS` 方法对应的进行语法分析的 BNF 范式为 “`<binoprhs> ::= ('+'|'-'|'*'|'/' <primary>)*`”。

```

static std::unique_ptr<ExprAST> ParseBinOpRHS(int ExprPrec, std::unique_ptr<ExprAST> LHS)
{
    while(true) {
        // 获取当前位置的运算符优先级
        int TokPrec = Lexer::GetTokPrecedence();
        // 当满足 TokPrec == -1 或 TokPrec == ExprPrec-1 的时候跳出循环
        if (TokPrec < ExprPrec) {
            return LHS;
        }
    }
}

```

```

// 获取当前位置的运算符字符
int BinOp = Lexer::CurTok;

// 获取下一个操作数
Lexer::GetNextToken();
// 获取下一个操作数对应的 AST 结构
auto RHS = ParsePrimary();
if (!RHS) {
    return nullptr;
}
// 获取下一个操作数右侧的运算符优先级
int NextPrec = Lexer::GetTokPrecedence();
// 与上一个运算符的优先级进行比较, 若后者较大, 则进行递归
if (TokPrec < NextPrec) {
    // 此处 “TokPrec + 1” 是为了处理连续两个相同运算符的情况
    RHS = ParseBinOpRHS(TokPrec + 1, std::move(RHS));
    if (!RHS) {
        return nullptr;
    }
}
// 构建当前已确定左值部分的 AST 结构
LHS = llvm::make_unique<BinaryExprAST>(BinOp, std::move(LHS), std::move(RHS));
}
}

```

到这里, 语法分析器中用于处理数学表达式类型语法结构的几个基本的 BNF 解析方法便编写完成了。当调用 `ParseExpression` 方法时, 语法分析器便会从源代码 I/O 流的当前位置处, 直接按照数学表达式的语法组成结构来进行相应的语法分析。

限于篇幅, 对于 `Cinderella` 语言中另一个重要的语法特性“函数定义”对应的语法分析器实现流程, 这里不再介绍, 读者可以按照同样的思路来自行分析。不过值得注意的是, 在定义与函数相关的 AST 节点类代码中, `PrototypeAST` 节点类主要用于定义函数的声明(函数名及形式参数)部分; `FunctionAST` 才是真正用于定义函数体的 AST 节点类。参照函数定义语法对应的 BNF 范式 “<definition> ::= 'def' <prototype> <expression>”, 读者可能会理解得更加深刻。

4.2.6 生成 LLVM-IR 代码

在将从外部输入给编译器的源代码转换成 AST 结构之后, 接下来便可以开始与 LLVM 工

具链进行对接了。如图 4-37 所示，在这一步中，我们将会通过为每一个 AST 节点类实现其各自对应的“codegen()”接口这种方式，来让 LLVM 帮助我们生成整个源代码 AST 结构对应的 LLVM-IR 中间代码。

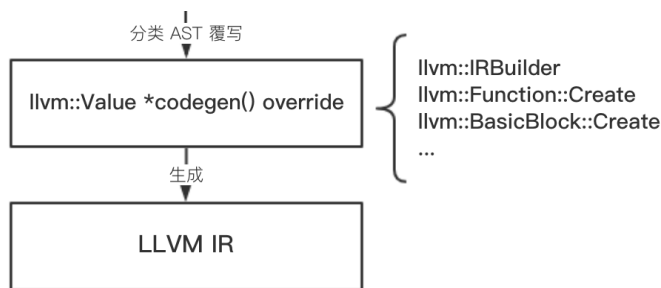


图4-37 从AST到LLVM-IR中间代码的部分链路

在开始为各子 AST 节点类实现对应的 IR 代码生成函数之前，我们还需要在当前的代码环境中创建如下几个全局静态变量。其中，第一个被声明为“llvm::LLVMContext”类型的变量是一个用于存放如“常量值列表”等大量 LLVM 核心数据结构的不透明对象。在这里并不需要关注该结构的内部实现细节，只需要通过相应的 API 接口与其进行交互即可；第二个类型为“llvm::IRBuilder”的变量主要用于辅助并简化从源代码 AST 结构生成对应 LLVM-IR 中间代码的过程，在该变量对应的类实体中定义了多个用于创建 LLVM-IR 语法元素的公共方法；第三个被声明为“std::map”类型的变量主要用于记录在当前作用域内定义的所有变量，以及与之相对应的 LLVM-IR 值对象，本质上相当于一个符号表结构。最后一个被定义为“llvm::Module”类型的变量在其内部存储了与当前 LLVM 模块相关的所有信息。在这里“模块类型”是所有其他 LLVM-IR 类型对象的顶级容器。每一个模块都将直接包含与当前源代码相关的全局变量列表、函数列表、模块依赖的函数库列表（或其他模块）、符号表，以及与编译器后端相关的目标平台特性等各种数据。

```

llvm::LLVMContext TheContext;
llvm::IRBuilder<> Builder(TheContext);
std::map<std::string, llvm::Value*> NamedValues;
std::unique_ptr<llvm::Module> TheModule;
  
```

在创建好上面这些具有特定功能的全局静态变量之后，我们便可以着手实现各个 AST 子类内部的 codegen() 方法了。首先需要在所有 AST 节点类的父类 ExprAST 中，添加同名的 codegen 纯虚函数作为其所有子类的统一实现接口。需要注意的是，该接口对应函数实现的返回值必须是 llvm::Value 类型的指针。这是由于 LLVM 在其内部使用了 llvm::Value 这个类来作为所有 IR 值类型的统一基类。

```
ExprAST.h
```

```
...
virtual llvm::Value *codegen() = 0;
...
```

接下来，将为各个 AST 节点类实现其各自的 `codegen()` 接口。这里我们仍然从终结符（数字/字符串标识符）对应的 AST 节点类开始改写。在下面这段代码中，我们为 `NumberExprAST` 子类实现了在其父类中定义的 `codegen()` 接口。这里直接通过 `llvm::ConstantFP` 类结构创建了 LLVM 内部的一个浮点数常量并将其返回。实际上，LLVM 内部的所有常量都只保留一份实体数据，并且该实体还会在整个模块内的各个地方进行共享。因此，在 `llvm::ConstantFP` 类中，名为 `get()` 的静态方法在实际执行时会首先从在当前模块 `TheContext` 对象中维护的常量池内来查找对应常量值，若找到则直接将其返回；否则，会先创建，然后再将其返回。

```
llvm::Value *NumberExprAST::codegen() {
    // 从常量池中查找对应常量
    return llvm::ConstantFP::get(TheContext, llvm::APFloat(Val));
}
```

接下来我们来实现“字符串标识符”终结符对应 AST 类结构中的 `codegen()` 接口。实际上，在该版本 `Cinderella` 语言中，字符串标识符仅用于表示函数定义中的“函数名”及“形式参数”这两种语法元素，因此我们直接从名为“`NamedValues`”的全局符号表中查找该标识符对应的 LLVM-IR 实体对象。而创建这些 LLVM-IR 实体的操作则会被放到函数定义 AST 节点类所对应的 `codegen()` 接口实现中进行。（可以参考项目源代码 `FunctionAST.cc` 文件中的“`LLVMBinder::NamedValues[Arg.getName()] = &Arg;`”语句。该语句中的“`Arg`”参数来自函数原型 AST 对应的 LLVM-IR 对象。而该对象中则保存着对应函数名以及形参的符号信息。）

```
llvm::Value *VariableExprAST::codegen() {
    // 直接从符号表中将标识符对应的 LLVM-IR 实体返回
    llvm::Value *V = NamedValues[Name];
    return V;
}
```

最后，我们实现用于模拟数学表达式语法结构的 `BinaryExprAST` 子类其内部的 `codegen()` 接口。如下所示，这里用到了之前在全局环境中创建好的类型为“`llvm::IRBuilder`”、名称为“`Builder`”的变量。在该对象内部定义了多个用于创建 LLVM-IR 语法结构的方法。比如这里调用的 `CreateFAdd` 等方法便可直接用于创建各类数学表达式对应的 LLVM-IR 结构。

```
llvm::Value *BinaryExprAST::codegen() {
    // 调用该表达式的左值与右值分别实现的 codegen() 方法
    llvm::Value *L = LHS->codegen();
```

```

llvm::Value *R = RHS->codegen();
if (!L || !R)
    return nullptr;
// 根据运算符选择生成不同的 LLVM-IR 结构
switch (Op) {
    case '+':
        return Builder.CreateFAdd(L, R, "add");
    case '-':
        return Builder.CreateFSub(L, R, "sub");
    case '*':
        return Builder.CreateFMul(L, R, "mul");
    case '/':
        return Builder.CreateFDiv(L, R, "div");
    default:
        return nullptr;
}
}

```

至此，与数学表达式语法特性相关的 AST 节点类其内部的 `codegen()` 方法已经全部实现完毕。而对于与函数定义相关的 AST 节点类，则可以通过类似的方式来实现其各个语法元素内部对应的 `codegen()` 方法。比如对于 `PrototypeAST` 类中关于函数声明部分的 LLVM-IR 代码生成过程，可以参考如图 4-38 所示的流程图来进行实现。

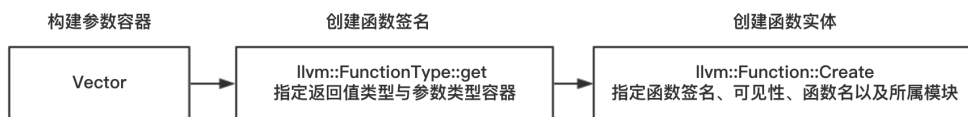


图4-38 PrototypeAST类中的codegen()方法实现流程

同理，对于函数实体类 `FunctionAST`，也可以按照如图 4-39 所示的 LLVM-IR 构造流程来实现其 `codegen()` 方法。

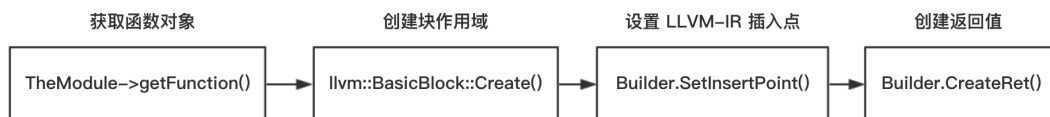


图4-39 FunctionAST类中的codegen()方法实现流程

在函数体定义的最后阶段，我们还调用了 LLVM 工具链提供的用于检测函数定义完整性的方法 `llvm::verifyFunction`，来验证函数定义是否存在问题。关于这部分语言特性的具体实现细节，读者可以参考 Github 上的完整源代码来进一步理解。读到这里，读者可能已经发现，基于 LLVM

工具链进行的一系列如从 AST 到 LLVM-IR 转换等过程，远比之前实现词法分析器和语法分析器的过程要简单得多。不仅如此，流程化的操作步骤也使得我们可以更加专注地设计语言本身的细节，而不用过多考虑在编译器实现上可能会出现的问题。

4.2.7 链接优化器

在上一节中，我们将源代码对应的 AST 结构转换成了 LLVM-IR 中间代码。按照正常的编译器链路流程，接下来需要对这些中间代码进行一定程度的优化，以便编译器后端能够输出体积更小、运行效率更高的二进制目标代码。同样的，基于 LLVM 工具链为编译器添加代码优化器的过程也是十分简单的。但首先需要了解的是，在 LLVM 中用于进行中间代码格式转换和优化的“Pass”系统是如何工作的。

在 LLVM 体系中，每一个“Pass”都分别对应于一个特定的代码处理函数，而整个“Pass”系统则由统一的 PassManager 对象来进行管理。与我们在 Linux 命令行中使用的管道操作符类似，每一个“Pass”函数都独立对应于一种特定的代码格式转换过程，我们可以在编译器源代码中以类似于管道的形式为 LLVM-IR 源串行地接入多个“Pass”处理函数。不同“Pass”节点函数的不同处理顺序会对源代码产生不同的影响，但同时也为开发者提供了足够高的灵活性，使其可以动态地选择在每一个代码生成阶段想要进行的优化流程。

如下所示，为了能够在编译器中使用 LLVM 提供的各类“Pass”优化器方法，首先需要创建一个用于管理“Pass”节点函数的 PassManager 对象。在 Cinderella 中，所有的基本表达式类型都需要被放置在函数结构中进行使用（函数共享库），因此这里选择 FunctionPassManager 子类来作为专门用于 LLVM-IR 函数体优化的“Pass”函数管理对象。

```
static std::unique_ptr<llvm::legacy::FunctionPassManager> TheFPM;
TheFPM = llvm::make_unique<llvm::legacy::FunctionPassManager>(TheModule.get());
```

可以看到，这里在初始化 FunctionPassManager 对象时传入了当前模块对应的 TheModule 对象作为参数。接下来，我们便可以在该对象的基础上为之前生成的函数体添加相应的“Pass”优化器方法，如图 4-40 所示。

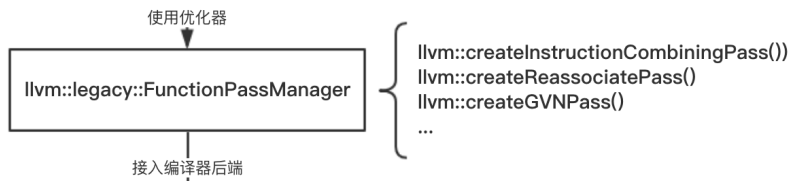


图4-40 LLVM优化器部分链路

这部分代码如下。

```
// 进行 Peephole 优化
TheFPM->add(llvm::createInstructionCombiningPass());
// 重关联表达式优化
TheFPM->add(llvm::createReassociatePass());
// 消除公共子表达式
TheFPM->add(llvm::createGVNPass());
// 简化 CFG 图
TheFPM->add(llvm::createCFGSimplificationPass());
```

可以看到，这里让 `FunctionPassManager` 对象通过其自身的 `add` 方法连接了多个具有不同功能的“Pass”优化器函数。随后，我们可以通过调用该 `PassManager` 对象中的 `run` 方法来启动这些优化器并触发优化过程。为了让用于定义函数体的 `FunctionAST` 节点能够在生成 LLVM-IR 代码时使用这些优化器来优化函数体的中间代码，我们可以在该节点类内部对应于 `codegen()` 方法的实现过程的最后添加如下一行代码。

```
// 传入 TheFunction 类型的函数体指针
TheFPM->run(*TheFunction);
```

在这行代码中，我们通过调用全局 `FunctionPassManager` 对象内部的 `run` 方法来绑定需要优化的目标函数体，并同时启动了优化过程。整个流程就是这样直观、简捷，优化器本身作为“黑盒”我们完全不用关心其内部的具体实现细节，只需要了解各优化器方案的优化策略即可直接“开箱使用”。

至此，我们便完成了整个编译器前端涉及的所有工作（词法分析、语法分析、生成 LLVM-IR 中间代码及链接优化器）。接下来，我们将再次借助 LLVM 工具链的力量来完成最后阶段的从 LLVM-IR 到目标平台代码的编译过程。

4.2.8 编译到目标代码

由于 LLVM 工具链本身支持多平台的交叉编译过程，因此，我们可以在当前单一平台的环境下编译生成对应不同平台架构的二进制目标代码。如图 4-41 所示，接入 LLVM 预置编译器后端的流程也十分清晰和简单，这里将为 Cinderella 语言编译器添加两种目标平台代码的后端生成器。其中一种是当前编译器运行所在平台；另一种是 WebAssembly 的虚拟平台。

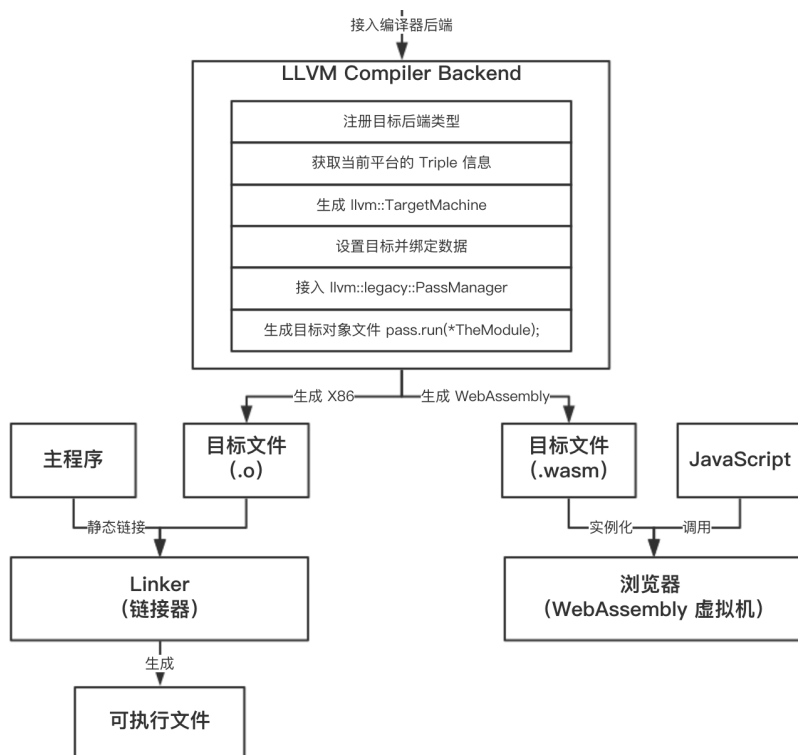


图4-41 编译到目标代码部分链路

根据图 4-41 所示的流程,首先需要注册将要使用的目标后端类型。通过该步骤可以让 LLVM 提前完成相应的后端初始化工作,以便后面可以直接调用这些可用目标平台的后端代码生成器。该步骤对应代码如下。为了简化实现过程,这里我们直接注册了在 LLVM 工具链内部所有可用的目标平台类型。

```

llvm::InitializeAllTargetInfos();
llvm::InitializeAllTargets();
llvm::InitializeAllTargetMCs();
llvm::InitializeAllAsmParsers();
llvm::InitializeAllAsmPrinters();

```

我们在之前的内容中介绍过, LLVM 在其内部会使用一种名为“Triple 字符串”的字符序列来表示某一具体平台架构的类型。那么,在接下来的步骤中,我们将会使用该字符串来查找需要编译生成对应后端代码的目标平台类型,并通过相应的方法返回表示该平台的一个“Target”对象指针。如下面代码所示,这里通过 `llvm::TargetRegistry::lookupTarget` 方法来查找需要的目标平台对象是否存在。由于需要创建分别对应于当前编译器运行平台及 WebAssembly 虚拟

平台的“Target”对象，因此在编译器的上层交互逻辑中可以让用户来选择目标平台类型。

在这里，当用户没有指定任何对应具体目标平台类型的参数时，我们会直接通过 `llvm::sys::getDefaultTargetTriple` 方法来获取并创建当前平台所对应的“Target”对象；否则，将会从名为“ISAList”的 Map 容器中查找指定平台名称对应的“Triple 字符串”。需要注意的是，WebAssembly 虚拟平台具有两种形式的 Triple 字符串，它们分别对应于两种不同的 ABI 类型。

其中，第一种名为“wasm32-unknown-unknown-elf”的 Triple 字符串主要应用在 Binaryen 工具链的实现中。通过该 Triple 字符串，LLVM 可以将 LLVM-IR 代码编译成以“.s”为后缀的 Wasm 可读汇编代码文件。该格式的代码文件一般会被应用在 Binaryen 内部的 s2wasm 工具中。该工具会将汇编代码文件描述的 Wasm 模块结构转译成 Binaryen-IR 中间代码，以便再次进行优化。因此，该 Triple 字符串显然并不符合我们需要直接生成 Wasm 二进制模块的编译器需求。

第二种名为“wasm32-unknown-unknown-wasm”的 Triple 字符串则与第一种恰好相反。基于该 Triple 字符串，LLVM 可以直接将经过优化的 LLVM-IR 中间代码编译成二进制格式的 Wasm 模块，而不需要任何中间过程。

```
std::map<std::string, std::string> ISAList = {
    // 使用 Triple 字符串 “wasm32-unknown-unknown-wasm”
    std::pair<std::string, std::string>("WASM", "wasm32-unknown-unknown-wasm")
};
...
std::string TargetTriple;
if (triple.size() == 0) {
    TargetTriple = llvm::sys::getDefaultTargetTriple();
} else {
    // WebAssembly Triple: "wasm32-unknown-unknown-wasm";
    TargetTriple = ISAList.find(triple)->second;
}
// 查找目标平台
auto Target = llvm::TargetRegistry::lookupTarget(TargetTriple, Error);
// 不存在则报错
if (!Target) {
    llvm::errs() << Error;
    return false;
}
```

上面这段代码主要用来检验用户指定的目标平台类型是否存在，如果不存在，则向用户上层抛出错误信息。

接下来，我们将真正进入创建后端目标平台对象的过程。如下面代码所示，这里主要通过 `createTargetMachine` 方法来创建目标虚拟机对象。该方法接收的第一个参数为对应目标平台的完整 Triple 字符串；第二个参数为该虚拟机的 CPU 类型，不同的 CPU 类型可能拥有不同的特殊功能属性，这里我们选择了最基本的“generic”通用类型；第三个参数用于指定当前目标虚拟机需要支持的额外功能特性，比如指定虚拟机是否需要支持可用于实现 SIMD 技术的 SSE 指令集；最后两个参数分别用于设置目标代码生成规则上的一些可选属性，以及与符号重定向策略相关的一些基本信息，这里全部使用默认值。

```
auto CPU = "generic";
auto Features = "";

llvm::TargetOptions opt;
auto RM = llvm::Optional<llvm::Reloc::Model>();
llvm::TargetMachine* TheTargetMachine = Target->createTargetMachine(TargetTriple, CPU,
Features, opt, RM);

// 设置模块的目标平台与数据布局格式（可选，将有利于代码优化）
TheModule->setDataLayout(TheTargetMachine->createDataLayout());
TheModule->setTargetTriple(TargetTriple);
```

需要注意的是，上面代码中的最后两行并不是必需的。这里是为当前 LLVM 模块对象设置对应后端目标平台的 Triple 类型，以及目标机器的数据布局格式，这样做可以在一定程度上有利于各“Pass”优化器对 LLVM-IR 代码的优化过程。整个编译器后端的最后一步动作就是设置输出文件的格式，并使用我们之前设置好的目标虚拟机对象来生成对应的目标二进制文件。

如下面代码所示，这里通过 `addPassesToEmitFile` 方法将生成目标二进制文件这个任务放到了 `llvm::legacy::PassManager` 对象中。我们之前介绍过，LLVM 中的“Pass”节点不仅可以用于对 LLVM-IR 代码进行优化，而且还可用于对 LLVM-IR 代码进行其他类型的格式转换。这里可以将该 `PassManager` 对象理解为一个简单的任务队列结构，与之前绑定优化器的“Pass”方法类似，将生成目标二进制文件这个过程也当作一个“Pass”处理动作。在代码中设置了目标文件为 `llvm::TargetMachine::CGFT_ObjectFile` 类型的对象文件，并同时将该类型参数、目标文件的输出地址和 `PassManager` 对象一同作为参数传递给了 `addPassesToEmitFile` 方法。该方法在其内部会检验目标文件的类型是否正确，并同时生成目标文件这个“Pass”动作挂载到当前传入的 `PassManager` 对象上。当这一切都准备就绪后，我们便可以通过在该 LLVM 模块对象中调用 `PassManager` 对象的 `run` 方法来生成目标文件了。

```

// 传入的目标文件名
auto Filename = fileName;
std::error_code EC;
// 创建目标文件的输出流对象
llvm::raw_fd_ostream dest(Filename, EC, llvm::sys::fs::F_None);

if (EC) {
    llvm::errs() << "Could not open file: " << EC.message();
    return false;
}

llvm::legacy::PassManager pass;
// 设置生成文件类型（二进制对象文件）
auto FileType = llvm::TargetMachine::CGFT_ObjectFile;
// 将生成目标文件的“Pass”动作挂载到 PassManager 对象上
if (TheTargetMachine->addPassesToEmitFile(pass, dest, FileType)) {
    llvm::errs() << "TargetMachine can't emit a file of this type";
    return false;
}
// 依次执行 PassManager 对象上挂载的“Pass”动作（生成目标文件）
pass.run(*TheModule);

```

至此，Cinderella 语言的编译器核心部分就基本实现完成了。但是除了最核心部分的词法分析器、语法分析器、优化器和编译器后端，还需要实现的便是最上层用于处理源代码文件 I/O（读取源代码），以及与用户进行指令交互的部分逻辑功能。

4.2.9 整合 I/O 交互层

在整个编译器的实现链路中，有两个地方会涉及数据的 I/O 与交互功能，其中第一个地方就是直接与用户打交道的命令行环境。用户在运行编译器对应的二进制可执行程序时，需要在调用命令中加入一系列的编译器指令控制参数。比如用于指定编译目标平台类型的“-t”参数，以及用于指定目标文件名的“-o”参数。不仅如此，被放置在命令行语句最后的源代码文件路径也同样需要作为参数被编译器应用程序处理。第二个地方是对源代码文件的内容读取过程。这里为了实现方便，我们会一次性将源代码文件中的所有字符内容全部读取到当前应用程序的内存空间中，然后在内存中再逐步完成对源代码的词法分析、语法分析、中间代码优化，以及最后生成目标平台二进制文件等过程。下面我们将分别介绍这两部分功能的具体实现细节。

命令行交互

如下面代码所示，这里给出了命令行交互功能的最简单实现版本。本质上，在 C/C++ 语言

中，我们可以直接从“argv”这个二维字符数组中读入从命令行传递给应用程序的所有参数值。在这段代码中，我们使用了十分暴力的参数匹配方法，即直接按照“参数值 参数名”这样的形式，从输入到编译器的所有参数中依次匹配出对应的参数键值对。这些键值对随后会被存放到一个字典结构中，便于后续的参数使用。当然，关于更好的命令行交互方式可以参考 Github 中 Cinderella 项目的完整源代码，在那里我们构建了一个更为成熟、完善的命令匹配与帮助系统。

```
#include <iostream>
#include <map>
#include <string>
#include <vector>

using namespace std;

int main (int argc, char *argv[]) {
    map<string, string> params;
    vector<string> temp;
    int counter = 0;
    // 读入字符数组直到结束
    while(auto e = argv[counter]) {
        temp.push_back(string(e));
        if (counter != 0 && counter % 2 == 0) {
            // 参数名与参数值作为键值对被存入字典结构中
            params.insert(pair<string, string>(temp[counter - 1], temp[counter]));
        }
        counter += 1;
    }

    // 判断参数“-t”是否存在
    if (params.find("-t") != params.end()) {
        cout << "The target platform is: " << params.find("-t")->second << endl;
    }

    // 判断参数“-o”是否存在
    if (params.find("-o") != params.end()) {
        cout << "The output path is: " << params.find("-o")->second << endl;
    }

    // 读入最后的源文件路径参数
    cout << "The input source file is: " << temp[counter - 1] << endl;

    return 0;
}
```

源文件读取

还记得前面在构建词法分析器时，我们曾通过调用 `IOInterface::ReadCharacterSource` 方法来获取源代码中当前读取位置的下一个字符。事实上，该方法的本质其实就是从缓存有源代码文件内容的“buffer”数组中将对应索引位置处的字符直接返回。而在 `IOInterface` 类中，整型的静态成员属性 `bufferPointer` 就是用于维护源代码的当前读取索引位置的。当整个编译器开始运行时，我们首先会通过调用该类中的 `InitialBufferPayload` 方法来初始化这个存放着源代码文件内容的“buffer”数组。该方法的实现细节如下。

```
void IOInterface::InitialBufferPayload (const std::string &fileName) {
    std::ifstream in(fileName);
    CompileSourceName = fileName;
    char c;
    int counter = 0;

    if(in.is_open()) {
        while(in.good()) {
            in.get(c);
            // 将所有读入的字符存入数组中
            buffer[counter] = c;
            counter++;
        }
        buffer[counter] = EOF;
    }

    if(!in.eof() && in.fail()) {
        std::cout << "[Cinderella] Error reading " << CompileSourceName << std::endl;
    }

    in.close();
}
```

可以看到，该方法的实现过程十分简单，但也存在弊端。比如这里“buffer”数组的大小其实是一个固定值，当源代码的字符数量超过 1024 个时，方法便会向外抛出异常。另外，如果源代码的体积过大，一次性将其完全读入内存这种方式也并不可取，因为串行的文件读取及代码处理过程也会让编译器的编译效率大大降低。那么，应该怎样进行优化呢？这个问题留给读者来思考。

至此，一个可用于 Cinderella 语言的静态编译器就构建完成了。从整体上看，基于 LLVM 提供的优化器、编译器后端等组件，可以省去我们的大量工作，并将更多的精力放到语言本身

的细节设计上来。不仅如此，直接将 LLVM 工具链作为独立的命令行工具来使用，其本身也是十分灵活和方便的。

4.3 WAT

关于 WAT (WebAssembly Text Format) 这种文本格式，在前面的内容中我们多次提到过，并给出了很多示例代码。在本节中，我们将重新审视 WAT 的组成细节、应用特性及历史由来。

我们将从如下这段 C/C++ 代码开始说起。在这段代码中，首先创建了一个 `fib` 函数，并在该函数内实现了求斐波那契数列中给定参数 “`x`” 对应位置元素值的功能。

```
int fib (int x) {  
    if (x < 2) {  
        return 1;  
    } else {  
        return fib(x - 1) + fib(x - 2);  
    }  
}
```

接下来，我们可以通过 `WasmFiddle` 或其他类似的线上、线下编译平台，将上面的 C/C++ 源代码文件编译为 WAT 格式的文本文件，该文件内的代码如下。

```
(module  
  (table 0 anyfunc)  
  (memory $0 1)  
  (export "memory" (memory $0))  
  (export "fib" (func $fib))  
  (func $fib (param $0 i32) (result i32)  
    (local $1 i32)  
    (set_local $1  
      (i32.const 1)  
    )  
    (block $label$0  
      (br_if $label$0  
        (i32.lt_s  
          (get_local $0)  
          (i32.const 2)  
        )  
      )  
    )  
    (set_local $1  
      (i32.const 1)  
    )  
  )  
)
```

```
)
(loop $label$1
  (set_local $1
    (i32.add
      (call $fib
        (i32.add
          (get_local $0)
          (i32.const -1)
        )
      )
    )
    (get_local $1)
  )
)
(set_local $0
  (i32.add
    (get_local $0)
    (i32.const -2)
  )
)
(br_if $label$1
  (i32.gt_s
    (get_local $0)
    (i32.const 1)
  )
)
)
)
)
(get_local $1)
)
)
```

这里给出官方对 WAT 格式的定义：WAT 是一种与 Wasm 二进制格式等效的，可以用于对 WebAssembly 模块及其包含的所有定义（即模块内容）进行编码的一种文本格式。这种格式使用“S-表达式”来表达模块的定义过程，并同时允许将函数体中的代码以线性的方式进行表示。该格式可以被与 Wasm 相关的编译工具使用，并且 WAT 格式也会代替二进制的 Wasm 编码来作为在浏览器中进行 WebAssembly 模块调试时对应的源代码文本。

4.3.1 S-表达式

既然说 WAT 文本格式是基于“S-表达式”来表达模块的定义过程的，那么什么是“S-表达式”呢？“S-表达式（S-expression）”是一种以人类可读的文本形式来表达半结构化数据的语法

约定，其名称中的大写字母“S”对应于“符号”(Symbol)一词。比如对于“ $2 * (3 + 4)$ ”这样一个以中缀形式表示的基本数学表达式，则可以通过“S-表达式”将其改写成如下形式。

```
( * 2 ( + 3 4 ) )
```

可以看到，在这个“S-表达式”中，我们使用小括号“ $()$ ”来定义每一个子表达式，并且子表达式与子表达式之间还可以根据相应的语法规则来相互嵌套，这从总体上构建出了 AST（抽象语法树）的结构。除此之外，“S-表达式”还会使用前缀形式来展现操作数与操作符之间的从属关系。比如对于上述表达式中最内层的子表达式“ $(+ 3 4)$ ”，可以很直观地看到位于小括号内的表达式内容是以 PN（波兰表达式）的形式来组织和展现的。如图 4-42 所示，从这个例子中可以更加清楚地看到，“S-表达式”是如何展现一个数学表达式对应 AST 结构中各节点的对应关系的。

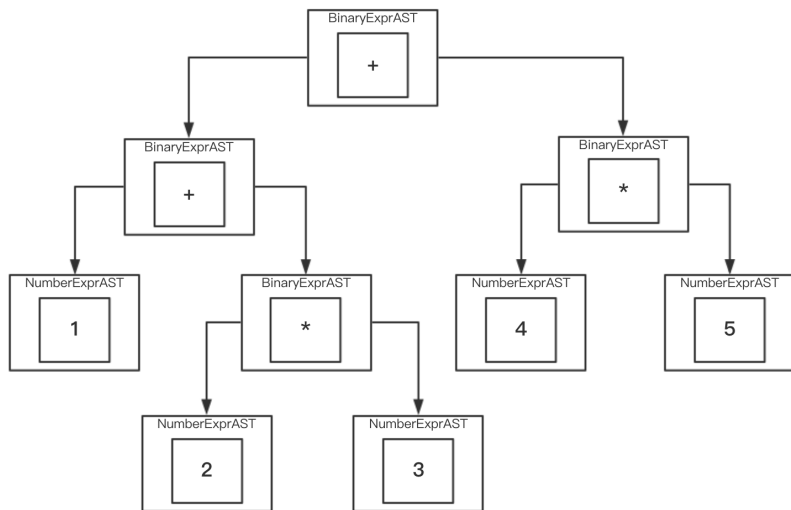


图4-42 一个数学表达式对应的AST（抽象语法树）结构

接下来，我们将这个 AST 结构对应的数学表达式“ $1 + 2 * 3 + 4 * 5$ ”分别转换为“S-表达式”与 WAT 两种形式。下面首先给出的是“S-表达式”形式。

```
(+
  (+ 1
    (* 2 3)
  )
  (* 4 5)
)
```

这里将“S-表达式”进行了换行、缩进等格式化处理，以便读者能够更加直观地看清整个表达式的组成细节。对照着图 4-42，我们从里向外来解析该“S-表达式”的结构。首先，位于

整个表达式最内层的子表达式 “ $(* 2 3)$ ” 正对应着我们从下向上来解析 AST 时所遇到的第一棵子树的结构。接下来，该子树结构又被作为参数传递给上一层的 “ $(+ 1)$ ” 子表达式，这里也与 AST 中对应位置的树和子树关系相对应。因此，我们可以说 “S-表达式” 形式可以在一定程度上描述应用程序源代码的 AST 结构。下面给出的是该表达式对应的 WAT 代码形式。

```
(func $calculate(; 0 ;) (result i32)
  (i32.add
    (i32.add
      (i32.const 1)
      (i32.mul
        (i32.const 2) (i32.const 3)
      )
    )
  )
  (i32.mul
    (i32.const 4) (i32.const 5)
  )
)
```

这里我们直接构建了一个名为 `calculate` 的函数，在函数体内计算了数学表达式 “ $1 + 2 * 3 + 4 * 5$ ” 的最终值，并将计算结果返回。抛开函数定义部分对应的 WAT 代码不看，如果将组成数学表达式部分的 WAT 代码与之前的 “S-表达式” 代码进行对比，就会发现两者之间唯一的区别是，在 WAT 代码中需要使用 WebAssembly 标准中的虚拟指令来代替实际表达式中的各类运算符与操作数。除此之外，在语法的表达结构和表现形式上没有任何区别。

4.3.2 WAT/Wasm 与 Binary-AST

在上一节中，我们介绍了 “S-表达式” 与 WAT 两者在代码表现形式上的异同之处。其实从本质上说，WebAssembly 模块的二进制代码格式在组成结构上与 WAT 格式和 “S-表达式” 也有相似之处。接下来，我们仍以上面的数学表达式为例，来对比 WAT 代码、Wasm 二进制模块代码和 “S-表达式” 三者之间的不同表现形式及对应关系。

在这里，我们可以通过 `wat2wasm` 等在线编译平台，直接将上一节中得到的数学表达式的 WAT 代码编译成对应的独立 Wasm 二进制模块。待编译完成后，我们可以通过 `hexdump` 命令将该模块中的内容以十六进制字面量数据的形式打印出来。下面给出的是模块中对应函数 `calculate` 的函数体定义部分内容。

```
41 02 41 03 6c 41 01 6a 41 04 41 05 6c 6a
```

我们之前介绍过，在 WebAssembly 标准中定义的每一个虚拟指令都有一个与之相对应的二进制 OpCode，在编译 Wasm 模块时，这些二进制指令代码会以线性的组织方式被写入模块的二进制文件中。因此，对于上面所给出的十六进制数据，其中的部分单字节数据将对应特定的 Wasm 虚拟指令。现在我们为这段数据中的每一个字节数据都添加相应的注释说明，相信这样读者会对 Wasm 模块的二进制代码组成结构有更加深刻的理解。

```

41 02      # i32.const 02
41 03      # i32.const 03
6c         # i32.mul
41 01      # i32.const 01
6a         # i32.add
41 04      # i32.const 04
41 05      # i32.const 05
6c         # i32.mul
6a         # i32.add

```

如上所示，我们将这段二进制（以十六进制表示）代码中每一个具有独立语义的虚拟指令连同其操作数均放置在单独的一行。由于 WebAssembly 虚拟机是基于栈结构实现的，因此在函数被实际调用时，虚拟机会由上至下依次执行上面各行二进制 OpCode 所对应虚拟指令的各种实际动作（如创建常量、定义函数结构，以及执行相关运算操作等）。整个二进制代码的解析与执行流程如图 4-43 所示。

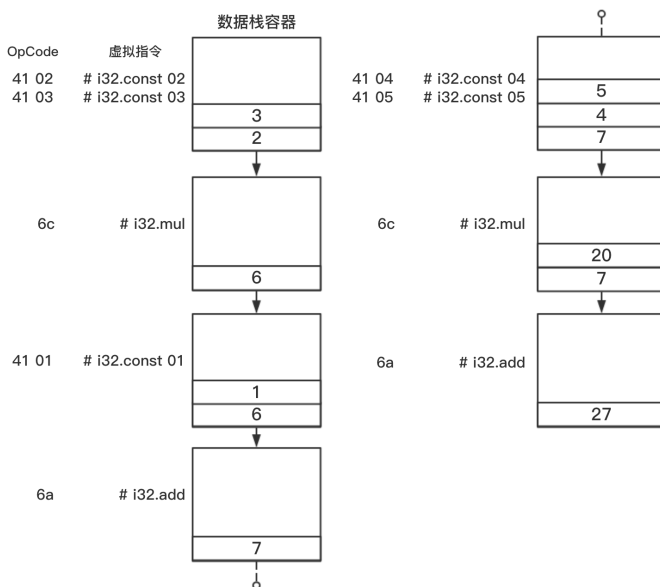


图4-43 上述二进制OpCode代码的宏观解析与执行流程

从图 4-43 中可以看到，当上述二进制代码执行时，所对应的数据栈容器中的数据变化过程。WebAssembly 虚拟机会线性地从上至下依次执行每一行 OpCode 对应的指令动作。与 WAT 不同的是，WAT 描述的是模块代码在语法层面的组成结构；而 Wasm 二进制文件内的 OpCode 则描述的是模块在实际执行时的线性指令组成结构。因此，从某种层面来讲，我们也可以称 Wasm 二进制文件内的 OpCode 代码是一种 Binary-AST 结构，或者更准确地将其称为“Linear Binary AST”；而 WAT 则对应着 Text-AST 这种可读文本类型的 AST 结构。

4.3.3 其他与设计原则

值得注意的是，在 WebAssembly 技术刚出现的那段时间里，我们经常会使用“.wast”来作为 Wasm 可读文本代码文件的文件名后缀名。但事实上，以“.wast”为后缀的文件通常是“.wat”文件的一个超集。即在该文件中，不仅可以包含有与模块定义相关的代码，而且还可以包含一些基于语法扩展定义的诸如“断言”等与测试相关的代码结构。而这部分语法结构却并不属于 WebAssembly 标准的一部分。相反的，以“.wat”为后缀的文件则只允许包含与 Wasm 模块本身语义相关的语法结构，并且在一个文件中也只能定义单一的模块结构。

从设计原则上看，Wasm 可读文本格式(.wat)在创建之初是为了满足以下几点要求。

- 希望能够有一种更加自然的方式在 Web 浏览器中展示某个 Wasm 模块的源代码语法结构，并且该方式必须具有一定的可读性。
- 可以被方便地应用于任何与 Wasm 相关的开发工具（如汇编器、调试器、优化器等）中。
- 在需要时可以直接编写该格式的 Wasm 模块代码，从而省略从 C/C++ 等上层高级语言到 Wasm 模块的编译过程。

至此，我们就介绍完了 WAT 文本格式的语法表达形式、它与“S-表达式”的对应关系，以及 Wasm 二进制 OpCode 代码的执行流程等内容。相信，如果你此刻回过头去重温我们在本书之前内容中给出的一系列 WAT 代码片段，将会有不一样的解读方式。关于 WAT 文本代码各组成部分对应的具体语法结构，可以参考 WebAssembly 官方网站上给出的标准说明。

第 5 章

Emscripten 基础应用

提示：本章使用的 emsdk 工具包是 1.38.0 版本。

在本书的前几章内容中，我们大量介绍了 WebAssembly 的核心技术原理、一些重要周边技术的底层运作细节，以及对 LLVM 工具链的深度应用。还记得我们曾在本书第 1 章中尝试通过 Emscripten 工具链构建了一个完整的 Wasm 应用。而从本章开始，我们将系统地介绍 Emscripten 工具链为我们提供的各种常用功能特性对应的使用方法及基本实现原理。借助 Emscripten 的强大功能，我们可以方便地利用 Wasm 技术来构建各类丰富的上层 Web 应用。

5.1 利器——Emscripten 工具链

本节将介绍 Emscripten 工具链的发展历史、组成结构、本地环境的安装与配置过程，并回忆基于 Emscripten 从零开始编写并构建一个完整 Wasm/ASM.js 应用的基本流程。

5.1.1 Emscripten 发展历史

Emscripten 是由 Mozilla 研究员 Alon Zakai 于 2010 年着手开发的一套基于 LLVM 构建的编译器工具链，通过该工具链我们可以将基于 C/C++ 语言编写的传统应用程序源代码轻松地转译为符合 ASM.js 标准特性的 JavaScript 代码（仅需要少量代码修改），并且这些 JavaScript 代码可以直接被 Web 浏览器解释和执行。不仅如此，由于 Emscripten 基于 LLVM 开发的特性，事实上所有可以被转译为 LLVM-IR 中间代码的编译器前端语言，都可以借助 Emscripten 工具链被直接转译成相对应的 JavaScript 代码，这样我们便可以方便地将基于不同源语言编写的应用程序快速无痛地移植并运行在 Web 平台上（这里以 ASM.js 的形式）。

但随着 ASM.js 存在的问题不断显现，以及 WebAssembly 技术的快速发展，Emscripten 工具链只支持转译到 ASM.js 代码这一特性便显得十分尴尬。虽然 ASM.js 技术其本身逐渐失去了竞争力，但作为一个从 C/C++ 到 JavaScript 的语言转译器，Emscripten 的整体实现已经相当成熟，其编译器后端生成的 ASM.js 代码具有非常高的可用性，并且在经过编译器的中间优化过程后，代码本身也具有了较高的运行效率。因此，Alon Zakai 于 2015 年 8 月底又创建了另外一个开源项目——Binaryen，当时该项目的主要目的就是用于将通过 Emscripten 工具链生成的 ASM.js 代码编译成对应的 WebAssembly 可读文本代码（WAT）或二进制模块。现如今，Emscripten 已经在其整个编译器链路中与 Binaryen 完全打通，这使得我们可以更加方便地直接通过 Emscripten 工具链来构建 Wasm 应用。

由于基于 LLVM 构建的 WebAssembly 后端出现时间较晚，因此，Emscripten 并没有在其编译器链路中直接使用我们在第 4 章中介绍的 wasm32-unknown-unknown-wasm 这个 Triple 字符串对应的目标后端类型。作为替代，Emscripten 则在其内部基于 LLVM 实现了一个从 LLVM-IR 中间代码编译到 ASM.js 代码的 JavaScript 后端。现在这个编译器后端的整体实现已经十分成熟。另外可喜的是，Emscripten 已经计划在其后续的版本中逐渐加入对 WebAssembly LLVM 后端的默认支持，如图 5-1 所示。

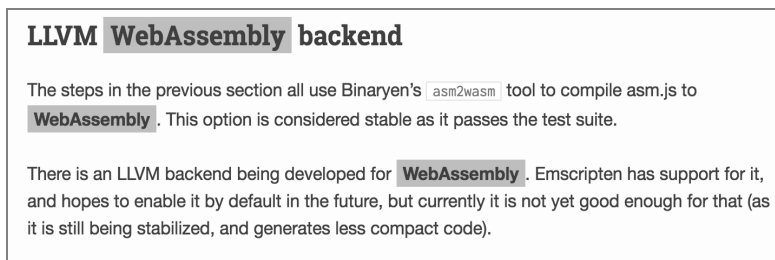


图5-1 Emscripten工具链计划默认启用WebAssembly的LLVM后端

实际上，任何可移植的 C/C++ 代码都可以通过 Emscripten 工具链被转译成对应的 JavaScript 代码而运行在 Web 浏览器中，其中包括使用 OpenGL/AL 技术来播放声音和处理图形的高性能 3D 游戏，以及基于 Qt 等 GUI 组件库编写的桌面应用程序等。到目前为止，Emscripten 已经成功地将一系列知名的代码库转换成相应的 JavaScript 版本，其中包括 CPython（基于 C 语言实现的 Python 解释器）、Poppler（基于 C/C++ 语言实现的 PDF 阅读器）和 Bullet Physics Engine（基于 C/C++ 语言实现的开源物理引擎）等大型项目，Unreal Engine 4 和 Unity3D 等商业游戏开发引擎也在其各自的软件开发工具包（SDK）中使用 Emscripten 工具链作为整体技术架构的一部分。如图 5-2 所示，Unity3D 引擎在其帮助文档中对如何使用基于浏览器的 WebGL 技术给出了说明。可以看到，这里引擎将使用 Emscripten 工具链来作为其上层代码的中间转译器。

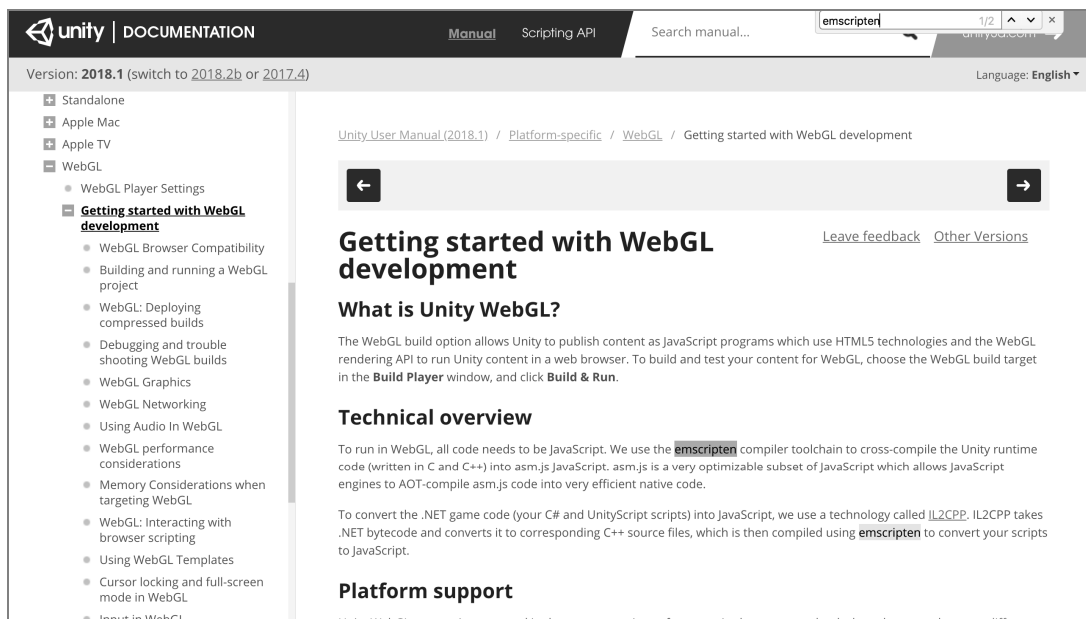


图5-2 在Unity3D引擎中使用WebGL技术

总的来说, Emscripten 作为一个十分成熟的 C/C++ 到 JavaScript 的转译器, 它帮助我们完善地处理了从 C/C++ 代码到 LLVM-IR 中间代码, 再到 ASM.js 代码的解析和转换过程。而且由于 ASM.js 与 WebAssembly 在部分技术特性上有着相似之处, 因此使得从 ASM.js 代码到 Wasm 模块的转译过程变得十分简单。我们经常会说: WebAssembly 今天的成就得益于昨天的 ASM.js 和 PNaCl。PNaCl 的出现, 让我们选择继续使用 LLVM-IR 来作为编译器链路的中间代码; 而 ASM.js 的出现, 则让我们能够快速地实现 WebAssembly 的 MVP 版本。

5.1.2 Emscripten 组成结构

从某种程度上讲, 虽然也可以称 Emscripten 是一种源到源的语言转译器, 但作为一套工具链, 它并非仅由单一的二进制可执行文件组成。与 LLVM 工具链类似, 在 Emscripten 中也同样提供了具有各种功能特性的 C/C++ 头文件、宏参数, 以及相应的命令行工具。其中的头文件和宏参数将被使用在应用程序对应的 C/C++ 源代码中, 用于指示 Emscripten 为源代码选择合适的编译流程, 并完成各种数据类型的转换。如图 5-3 所示为 Emscripten 编译器的链路结构。

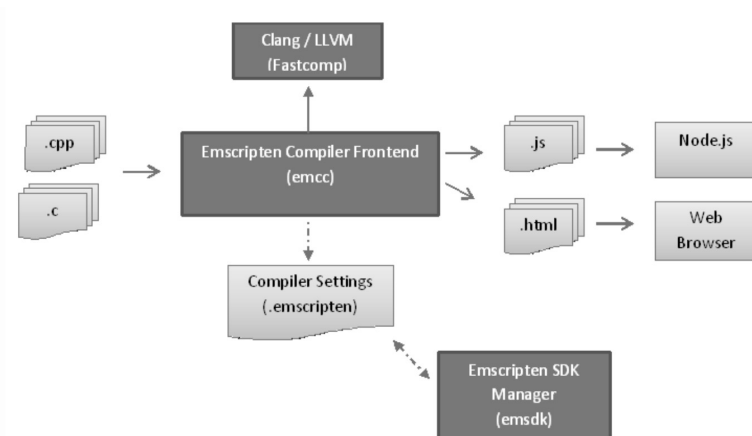


图5-3 Emscripten编译器的链路结构（图片来源于官网）

emcc (Emscripten Compiler Frontend)

emcc 是整个 Emscripten 工具链的核心编译器入口，其在内部直接复用了 Clang 编译器的前端部分，将输入的 C/C++源代码转换为 LLVM-IR 中间代码。因此，作为一个标准的 C/C++编译器前端，我们也可以选择使用如 GCC 等其他类型的成熟编译器来替换它。但需要注意的是，由于 Emscripten 本身是基于 LLVM-IR 来生成 ASM.js 代码的，因此这里我们需要选择类似 llvml-gcc 这类可以将源代码编译到 LLVM-IR 中间代码的编译器实现版本。不仅如此，emcc 也是 Emscripten 编译器链路的命令行工具入口。如图 5-4 所示，我们可以通过执行“emcc -v”命令来查看 Emscripten 编译器相关组件的当前版本信息。

```

→ emsdk git:(master) X emcc -v
emcc (Emscripten gcc/clang-like replacement + linker emulating GNU ld) 1.38.0
clang version 5.0.0 (emscripten 1.38.0 : 1.38.0)
Target: x86_64-apple-darwin16.6.0
Thread model: posix
InstalledDir: /Users/jason/Desktop/Repo/emscripten/clang/e1.38.0_64bit
INFO:root:(Emscripten: Running sanity checks)
  
```

图5-4 通过emcc来查看Emscripten编译器组件的相关版本信息

Fastcomp (Clang/LLVM)

Fastcomp 是 Emscripten 工具链中用于将中间状态 LLVM-IR 二进制代码转译为 ASM.js 代码的 JavaScript 编译器后端。该后端是完全基于 LLVM 工具链的后端开发标准构建的，因此可以在最大程度上保证从 LLVM-IR 生成目标代码的效率。

emsdk (Emscripten SDK Manager)

emscripten 是用于管理 Emscripten 工具链当前组件版本的开发工具包。其在内部同时管理并整

合了工具链多个子集工具（比如 Binaryen、Clang、Fastcomp 编译器后端等）之间的功能依赖关系。我们也将通过 emsdk 来配置 Emscripten 的开发环境。

~/.emscripten (Compiler Settings)

.emscripten 是一个具有特殊功能的 ASCII 文本文件，该文件被 emsdk 默认放置在当前用户的家目录（~）中。在通过 emsdk 搭建 Emscripten 开发环境的过程中，emsdk 会将当前编译开发环境需要用到的一些诸如编译链路的相关组件、各类命令行工具的所在位置、系统环境变量等信息写入该文件中。而在接下来的编译过程中，emcc 将会通过该文件来获取正确的工具链信息，并启动相应的源代码编译流程。

除此之外，还有如 JRE（Java Runtime Environment）等可选组件，读者可以根据实际需求自行选择安装。但是否安装这些附加组件并不会影响代码的编译过程。

5.1.3 Emscripten 下载、安装与配置

这里假设读者已经注册了 Github 账号，并且会使用最基本的 git 命令。接下来，我们可以按照如下步骤来下载、安装并配置 Emscripten 的编译开发环境。以下操作将在 MacOS 系统下进行。对于 Windows 和 Linux 系统，读者可以参考官方文档中相关说明进行操作。

（1）克隆源代码仓库到本地。

通过执行“git clone https://github.com/juj/emsdk.git”命令语句将远程的 emsdk 源代码仓库克隆至本地的某个位置。

（2）下载并安装最新版本的 SDK 工具。

首先通过“cd emsdk”命令进入 emsdk 源代码仓库所对应的文件夹内，然后再执行“./emsdk install sdk-1.38.0-64bit”命令语句，让 emsdk 自动帮助我们下载并安装版本为“1.38.0-64bit”的 Emscripten 开发包及其依赖的所有子集工具。（工具包版本号最后使用的 bit 位数字段依具体操作系统的类型而定，这里可以选择使用 32 或者 64。）

（3）激活工具。

当所有资源都准备就绪后，便可以通过“./emsdk activate latest”命令语句来激活当前的 SDK 工具包。在这里，所谓的“激活”实际上就是指将当前 Emscripten 工具链所需要用到的各种环境变量信息都写入.emscripten 配置文件中，以供如 emcc 等编译器工具使用。

(4) 初始化环境变量。

通过执行“`source ./emsdk_env.sh`”命令语句来连接当前命令行终端和 SDK 工具包内所有可用的命令行工具。该命令会将 `emsdk` 工具包中所有子集工具对应的可执行文件位置配置到当前用户的 `PATH` 系统变量中，并同时配置一些全局的环境变量。这样我们便可以在任意路径下使用 `emcc` 命令，而不用补全该命令所在位置的完整路径。

至此，整个 `emsdk` 工具包的部署过程就结束了。接下来，便可以通过 Emscripten 工具链来开发和编译一系列 WebAssembly 应用。

实质：基于 Python 构建的 Wrapper 脚本

在本章内容中，我们并不会将 `emsdk` 工具包内所有命令行工具的具体用法都介绍一遍。如果查看这些命令对应的实体文件，就会发现它们大多数都是基于 Python 语言编写的 Wrapper 脚本，而非传统的 ELF 二进制可执行文件。这些脚本在其内部通过一些事先封装好的“胶水”方法，将 `emsdk` 工具包内的三大核心组件，即 Binaryen 工具链、Clang 编译器和 Fastcomp 后端完美地结合在一起。这种“结合”带来的好处就是，`emsdk` 只需要向开发者暴露出一个名为“`emcc`”的核心命令，便可以完整地支持整个 WebAssembly 应用的构建过程。

如图 5-5 所示，我们来进一步观察对应于 `emcc` 命令的 `emsdk/emscripten/<version>/emcc.py` 文件内部的一些实现细节。可以看到，当在命令行中执行“`emcc -v`”命令时，实际上该命令在其脚本文件内部通过 `subprocess.call` 这种子进程调用方式间接地执行了“`clang -v`”命令，并且在执行该命令的前后又附加输出了与 `emcc` 本身相关的信息。

```

emscripten/emcc.py
Lines 373 to 378 in bf0bdc5
373     elif len(sys.argv) == 2 and sys.argv[1] == '-v': # -v with no inputs
374         # autoconf likes to see 'GNU' in the output to enable shared object support
375         print('emcc (Emscripten gcc/clang-like replacement + linker emulating GNU ld) %s'
376               code = subprocess.call([shared.CLANG, '-v'])
377         shared.check_sanity(force=True)
378         return code

```

图5-5 emcc.py文件内的部分代码实现细节

在命令行中分别执行“`clang -v`”和“`emcc -v`”命令后，两者向控制台中打印的消息内容如图 5-6 所示。可以看到，其中关于 Clang 编译器版本信息的部分内容是完全相同的，这与前面的源代码分析结果相符合。（这里调用的 `clang` 命令实际上是 `emsdk` 提供的专门用于 Emscripten 的经过源代码修改的 Clang 编译器。）

我们继续深入探索，在上述 `emcc.py` 文件中发现了如图 5-7 所示的一段代码，这段代码正

对应着 emcc 将 C/C++ 源代码编译成二进制 LLVM-IR 中间代码的过程。这里 emcc 在其内部通过 get_bitcode_args 方法为给定的源代码文件生成与之对应的 Clang 编译命令，该命令会与一些固定的必选参数（如用于指定 Clang 编译器生成 LLVM-IR 中间代码的“-emit-llvm”参数，以及用于指定目标文件输出位置及类型的“-o”参数等）进行拼接。当所有参数全部拼接完成后，emcc 便可以通过调用 execute 方法来执行这段指令语句。

```
→ emscripten git:(master) X clang -v
clang version 5.0.0 (emscripten 1.38.0 : 1.38.0)
Target: x86_64-apple-darwin16.6.0
Thread model: posix
InstalledDir: /Users/jason/Desktop/Repo/emsdk/clang/e1.38.0_64bit
→ emscripten git:(master) X emcc -v
emcc (Emscripten gcc/clang-like replacement + linker emulating GNU ld) 1.38.0
clang version 5.0.0 (emscripten 1.38.0 : 1.38.0)
Target: x86_64-apple-darwin16.6.0
Thread model: posix
InstalledDir: /Users/jason/Desktop/Repo/emsdk/clang/e1.38.0_64bit
INFO:root:(Emscripten: Running sanity checks)
```

图5-6 emcc和clang命令输出的版本信息

```
emscripten/emcc.py
Lines 1401 to 1409 in bf0bdc5

1401 def compile_source_file(i, input_file):
1402     logging.debug('compiling source file: ' + input_file)
1403     output_file = get_bitcode_file(input_file)
1404     temp_files.append((i, output_file))
1405     args = get_bitcode_args([input_file]) + ['-emit-llvm', '-c', '-o', output_file]
1406     logging.debug("running: " + ' '.join(shared.Building.doublequote_spaces(args)))
1407     execute(args) # let compiler frontend print directly, so colors are saved (PIPE
1408     if not os.path.exists(output_file):
1409         exit_with_error('compiler frontend failed to generate LLVM bitcode, halting')
```

图5-7 emcc编译器入口命令的内部实现细节

接着向下分析，我们发现了更多有意思的细节。如图 5-8 所示，可以看到，当 emcc 没有找到（启用）WebAssembly 汇编代码的原生 LLVM 后端（wasm32-unknown-unknown-elf）时，它便会选择使用 Binaryen 工具链中的 asm2wasm 工具，直接从 C/C++ 源代码对应的 ASM.js 代码来编译并生成相应的 Wasm 二进制模块。

```
emscripten/emcc.py
Lines 2324 to 2332 in bf0bdc5

2324 if not shared.Settings.WASM_BACKEND:
2325     if DEBUG:
2326         # save the asm.js input
2327         shared.safe_copy(asm_target, os.path.join(shared.get_emscripten_temp_dir(), o
2328 cmd = [os.path.join(binaryen_bin, 'asm2wasm'), asm_target, '--total-memory=' +
2329 if shared.Settings.BINARYEN_TRAP_MODE in ('js', 'clamp', 'allow'):
2330     cmd += ['--trap-mode=' + shared.Settings.BINARYEN_TRAP_MODE]
2331 else:
2332     exit_with_error('invalid BINARYEN_TRAP_MODE value: ' + shared.Settings.BINARY
```

图5-8 emcc编译器的不同编译决策

沿着 Wasm 模块的编译流程向下探索，最后在 emsdk/emscripten/<version>/tools/shared.py 文件中发现了如图 5-9 所示的 Fastcomp 编译器后端对应的 LLVM Triple 字符串（asmjs-unknown-emscripten）。但遗憾的是，与 WebAssembly 的原生 LLVM 后端一样，Fastcomp 也并没有被整合到 LLVM 的默认编译器后端支持列表中。

```

emscripten/tools/shared.py
Lines 888 to 890 in bf0bdc5

888     # Target choice.
889     ASMJS_TARGET = 'asmjs-unknown-emscripten'
890     WASM_TARGET = 'wasm32-unknown-unknown-elf'

```

图5-9 emcc命令在实现过程中使用到的不同Triple字符串

我们之前提到过，用于运行 Java 应用程序的 JRE（Java 运行时环境）也是 emsdk 工具包的可选组件之一，如图 5-10 所示的这段代码恰好验证了这个说法。可以看到，为了进一步压缩通过 Emscripten 工具链编译 Wasm 应用时生成的用于连接模块与 Web 浏览器的“胶水”JavaScript 脚本文件的体积大小，emsdk 在其工具集中引入了对“Google Closure Compiler（GCC）”这个超级 JavaScript 代码优化器的支持。但由于 GCC 本身是基于 Java 语言编写的，因此要运行它便需要由系统提供相应的 JRE 运行环境。

```

emscripten/tools/shared.py
Lines 432 to 441 in bf0bdc5

432     def check_closure_compiler():
433         try:
434             subprocess.call([JAVA, '-version'], stdout=PIPE, stderr=PIPE)
435         except:
436             logging.warning('java does not seem to exist, required for closure compiler, w
437             return False
438         if not os.path.exists(CLOSURE_COMPILER):
439             logging.warning('Closure compiler (%s) does not exist, check the paths in %s'
440             return False
441         return True

```

图5-10 Emscripten在其内部加入了对GCC代码优化器的支持

探索：.emscripten 配置文件的作用

我们之前介绍过，在.emscripten 文件中主要存放的是 emsdk 工具包中各类命令行工具在运行过程中需要使用到的各种环境变量参数。那么，这些参数究竟是以何种方式与这些编译工具相互配合协作的？下面给出了在.emscripten 文件中能够存放的一些基本字段内容。

```

import os
LLVM_ROOT = '<LLVM_ROOT>'
EMSCRIPTEN_NATIVE_OPTIMIZER = '<EMSCRIPTEN_NATIVE_OPTIMIZE>'
BINARYEN_ROOT = '<BINARYEN_ROOT>'
NODE_JS = '<NODE_JS>'

```

```

EMSCRIPTEN_ROOT = '<EMSCRIPTEN_ROOT>'
SPIDERMONKEY_ENGINE = '<SPIDERMONKEY_ENGINE>'
V8_ENGINE = '<V8_ENGINE>'
TEMP_DIR = '<TEMP_DIR>'
COMPILER_ENGINE = NODE_JS
JS_ENGINES = [NODE_JS]

```

虽然 `.emscripten` 文件并没有固定的文件名后缀，但是 `import` 关键字为我们提供了线索（上述第一行内容）。实际上，整个 `.emscripten` 文件确实是被当作一个基本的 Python 脚本文件来使用的。如图 5-11 所示，在 `emsdk/emscripten/<version>/tools/shared.py` 文件中，我们发现了 `emsdk` 将该文件中的内容读取出来并当作 Python 代码并解析执行的过程。当代码执行完毕后，在其内部声明的所有字段均会被作为全局变量而存在于整个 Python 运行时环境的最外层作用域中，这样 `emsdk` 内部的其他子集工具便可以在运行过程中直接使用这些变量参数的值。（大部分子集工具均是基于 Python 脚本构建的，并且其主流程也都包含了 `shared.py` 文件中定义的类似上述“读取配置文件信息”的公用代码逻辑，这也是该文件名“`shared`”所对应的含义。）

```

emscripten/tools/shared.py
Lines 245 to 253 in bf0bdc5

245     try:
246         config_text = open(CONFIG_FILE, 'r').read() if CONFIG_FILE else EM_CONFIG
247         exec(config_text)
248     except Exception as e:
249         logging.error('Error in evaluating %s (at %s): %s, text: %s' % (EM_CONFIG, CONFIG_FILE, e, config_text))
250         sys.exit(1)
251
252     # Returns a suggestion where current .emscripten config file might be located (if
253     # without a file, this hints to "default" location at ~/.emscripten)

```

图5-11 emsdk内部对~/.emscripten文件的使用细节代码

如果读者对上述 `.emscripten` 文件中这些变量参数的具体使用细节感兴趣，可以继续深入源代码进行研究。这里限于篇幅，不再详细介绍。

5.1.4 运行测试套件

`emsdk` 在其内部提供了一个非常全面的测试套件，该套件几乎覆盖了 Emscripten 工具链所能够提供给开发者的所有功能点。整个测试套件主要包含两个部分：第一部分为 `emsdk` 工具包的功能性测试脚本，该脚本会通过检查与 Emscripten 相关的所有功能是否可以被正常使用，来判断 `emsdk` 工具包的安装与部署是否正确；第二部分为相关的基准性能测试脚本，该脚本会将特定的 C/C++ 测试代码同时编译成 ASM.js、本地原生应用等多个平台代码，然后通过统计其各自多次执行花费的平均时间，来生成 Emscripten 工具链在各平台上对应各编译阶段的平均性能

统计报告。我们可以通过如下方式来使用这些测试脚本。

全覆盖测试

通过全覆盖测试，可以一次性执行 **emsdk** 工具包内预置的全部测试套件脚本。

```
python emsdk/emscripten/<version>/tests/runner.py
```

选择性测试

我们也可以选择性地单独运行 **emsdk** 测试套件中的某一项测试脚本。如下所示，我们单独执行了测试套件中的“基准性能测试”脚本。

```
python emsdk/emscripten/<version>/tests/runner.py benchmark
```

关于 **emsdk** 工具包测试套件的更多使用方法，读者可以参考 **emsdk/emscripten/<version>/test/runner.py** 文件中给出的注释信息。

5.1.5 编译到 ASM.js

本节主要介绍基于 **Emscripten** 工具链将 C/C++ 代码编译到 **ASM.js** 代码的具体流程，以及过程中需要注意的问题。

下面我们直接使用之前搭建好的开发编译环境来编写第一个基于 **Emscripten** 工具链构建的 **ASM.js** 应用。首先给出的是该应用对应的 C/C++ 源代码。

```
hello_emscriptem.cc
#include <iostream>

using namespace std;
// 定义一个求和函数
int add (int x, int y) {
    return x + y;
}

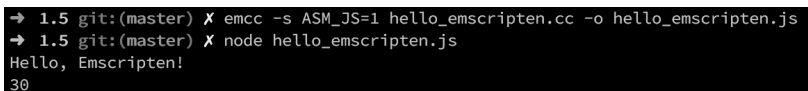
int main (int argc, char **argv) {
    // 调用 add 函数输出一行文本内容
    cout << "Hello, Emscripten! \n" << "Result: " << add(10, 20) << endl;
    return 0;
}
```

在上面的代码中，首先引入了名为“**iostream**”的 C++ 输入/输出标准库头文件，并在全局作用域内定义了一个求和函数 **add**。然后在主函数中编写了该 **ASM.js** 应用的主要流程代码。这

里通过 `cout` 对象向控制台打印了内容为 “Hello, Emscripten!” 的文本字符串，同时调用 `add` 函数并将其计算结果也打印出来。接下来，我们可以通过 Emscripten 工具链中的 `emcc` 编译器入口脚本（后面将简称为 “`emcc`”）来编译这段代码。对应的命令语句如下。

```
emcc hello_emscripten.cc
-s ASM_JS=1
-o hello_emscripten.js
```

在这段命令中，我们通过 “`-s`” 参数为 `emcc` 设置了 `ASM_JS` 选项标志，通过该标志可以告知编译器最终需要生成的目标代码类型。当上述命令执行完毕后，我们可以在源代码文件所在的文件夹内找到编译器生成的一个 JavaScript 脚本文件 `hello_emscripten.js`。现在我们直接通过 `node` 命令来执行该文件中的脚本代码，执行结果如图 5-12 所示。



```
→ 1.5 git:(master) X emcc -s ASM_JS=1 hello_emscripten.cc -o hello_emscripten.js
→ 1.5 git:(master) X node hello_emscripten.js
Hello, Emscripten!
30
```

图5-12 通过node命令运行上述ASM.js应用

可以看到，从源代码的编写到最后通过 Emscripten 工具链编译生成相应的 ASM.js 目标代码文件，在整个过程中并不需要完成任何复杂的参数配置步骤，而这均得益于 `emcc` 在其内部已经事先封装好了 `emsdk` 中各子集工具的详细调用流程。关于目标文件 `hello_emscripten.js` 的具体内容这里不进行解读，感兴趣的读者可以参考 ASM.js 标准进行深入分析。

在编译 ASM.js 代码时，以下一些信息值得我们关注。

“use asm” 与 “almost asm”

我们在第 1 章中介绍过，在定义一个 ASM.js 模块时，需要在该模块定义的开头使用 “`use asm`” 语句来告知浏览器我们当前正在定义的 JavaScript 函数其内部的所有代码均遵循 ASM.js 的语法和规则，即将整个函数视作一个标准的 ASM.js 模块。这样浏览器内部的 JavaScript 引擎才会按照 ASM.js 模块的数据分配与编译方式，来执行定义在该模块内的方法。

但实际上，浏览器对 ASM.js 代码的可优化条件非常严格。在模块定义中，一旦某些语法格式或边界条件没有满足浏览器对 ASM.js 模块定义的严格要求，整个 ASM.js 代码便会直接退化为正常的 JavaScript 代码——这部分代码虽然可以被 Web 浏览器执行，但却失去了很多只针对 ASM.js 模块的专有优化流程，代码的执行效率会大大降低。因此，在 Emscripten 工具链中，对于一些因含有特殊语法结构而无法被转译成标准 ASM.js 模块的 C/C++ 源代码，Emscripten 会将其对应的 ASM.js 模块类型声明标识改写为 “`almost asm`”，以表示该 ASM.js 模块不可被优化。

可选的 HTML 模式

在本节开头的 ASM.js 应用实例中，我们在 `emcc` 命令中直接将编译生成的目标文件类型设置成了以“.js”为后缀的 JavaScript 脚本文件，并通过 Node.js 以命令行的方式执行了该文件内的代码。实际上，除此之外，我们还可以选择在 Web 浏览器上运行该 ASM.js 应用。但并不需要再单独编写一个用来加载脚本文件的 HTML 文件，只需要将在编译命令语句中设置的目标文件后缀从“.js”更改为“.html”即可。这样，当 `emcc` 在编译源代码时，便会帮助我们同时生成该 ASM.js 应用对应的脚本文件，以及同名的用于加载该脚本文件的 HTML 文件。

在进行测试时，需要确保浏览器能够通过本地正在运行的 HTTP 服务器访问到上述生成的 HTML 文件。如图 5-13 所示为本节开头编写的 ASM.js 应用在 Web 浏览器中的实际运行结果。Emscripten 会在其当前构建的 HTML 文件中创建两个用于数据输出的区域，其中第一个是位于页面顶端的 Canvas 画布，在这里主要测试源代码中与 OpenGL 相关的部分功能；第二个是位于整个页面底端的可交互区域，在这里 Emscripten 模拟出一个只能进行数据输出的命令行控制台，可以看到，在源代码中通过 `cout` 对象输出的内容便被显示在这里。

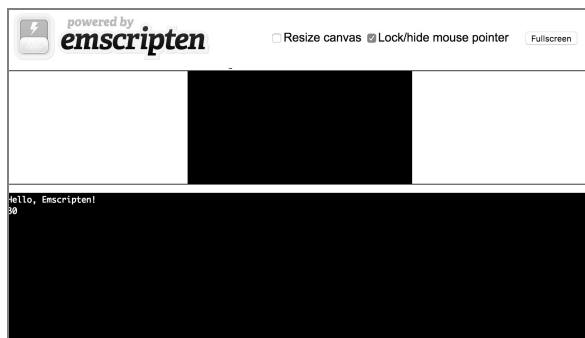


图5-13 本节开头的ASM.js应用在Web浏览器中的实际运行结果

a.out.js

在正常使用 `emcc` 来编译生成 ASM.js 代码时，一般需要在命令中通过“-o”参数来指定目标文件的存放位置、文件名和文件类型。但实际上，如果忘记在命令中指定与目标文件有关的参数，那么 Emscripten 会自动在当前目录下生成默认的名为“a.out.js”的目标文件。该文件与通过指定“-o”参数生成的目标文件在内容和使用方式上没有任何差别。

5.2 连接 C/C++与 WebAssembly

在本节中，将首先介绍在基于 Emscripten 工具链构建 Wasm 应用时可以使用的一些常用构

建类型, 以及这些应用类型之间的差别。然后介绍什么是 ERE (Emscripten Runtime Environment, Emscripten 运行时环境), 以及如何通过 ERE 在 WebAssembly 模块的内部与上层 JavaScript 代码进行交互和双向数据传递等内容。

5.2.1 构建类型

从本质上讲, 凡是能够被转译成 ASM.js 的 C/C++ 源代码, 都可以被 Emscripten 工具链再次间接转译成二进制形式的 Wasm 模块。但实际上, 单独使用 Wasm 模块并不能完全承载 C/C++ 源代码描述的所有功能。比如对于一个使用了 OpenGL 技术的 C/C++ 源程序, 在将其转译为 Wasm 目标代码时, Emscripten 会直接使用 Web 浏览器上的 WebGL 引擎来代替 OpenGL 的工作流程。但是由于这部分调用了 WebGL 接口的 JavaScript 代码其具体执行过程涉及浏览器本身的 WebGL 接口实现, 因此无法将这部分代码与独立的 Wasm 模块打包在一起。Emscripten 对此的解决办法是: 只将那部分不涉及浏览器层 API 接口, 仅具有纯计算和方法调用过程的代码打包并放置在统一的 Wasm 模块中。而对于那些需要与浏览器进行特殊交互或 JavaScript 接口调用的代码, 则将它们按照普通的 JavaScript 代码进行打包并交由浏览器执行。

因此根据 C/C++ 源程序的具体类型, 我们可以通过 Emscripten 工具链有选择地将其构建成为以下两种类型的 WebAssembly 应用。

Standalone

Standalone 类型的 Wasm 应用只适用于那些仅包含有纯计算和方法调用逻辑的 C/C++ 源程序, 即在源程序中不能含有任何涉及需要与浏览器 API 进行交互、发送远程 (HTTP/Socket) 请求, 以及与数据显示、输入等 I/O 相关的代码。Emscripten 在构建该类型 Wasm 应用时只会编译生成独立的 Wasm 二进制模块, 而不会帮助其构建任何用于连接该模块与上层 JavaScript 环境的脚本文件。下面我们将通过一个简单的示例来介绍该类型 Wasm 应用从源代码编写到编译生成目标应用的整个过程。

首先给出的是该应用中 Wasm 模块对应的 C/C++ 源代码。

```
emscripten-standalone.cc
```

```
#include <emscripten.h>

#ifdef __cplusplus
extern "C" {
#endif

// 函数定义, 利用 Emscripten 提供的宏防止函数被 DCE 处理掉
EMSCRIPTEN_KEEPALIVE int add (int x, int y) {
```

```

    return x + y;
}
#ifdef __cplusplus
}
#endif

```

这段代码十分简单。可以看到，在代码中我们定义了一个 `add` 函数，该函数将接收两个整型数据，并返回对它们进行求和计算后的结果。在这段代码中，引入了一个头文件 `emscripten.h`，该文件便是 Emscripten 工具链提供的用于连接 C/C++ 源代码与浏览器环境的“胶水工具”。在 C/C++ 源代码中，我们可以通过使用定义在该文件中的宏和函数来解决从 Native 转译到 Web 浏览器时所可能遇到的绝大部分跨平台问题。

比如，对于定义在该文件中的名为“EMSCRIPTEN_KEEPALIVE”的宏结构，如图 5-14 所示，Emscripten 官方文档对 EMSCRIPTEN_KEEPALIVE 宏的用法进行了详细描述。由于 emcc 在其内部调用了 Clang 的编译器前端，因此在 LLVM-IR 代码的分析与优化阶段，编译器会将源代码中没有被主函数（main）实际引用到的函数定义通过 DCE 过程将其从目标代码中移除。因此，对于只含有纯计算类函数定义的 C/C++ 源代码，我们便需要一种方法能够让编译器将源代码中没有被使用到的方法定义完整地保留到目标代码中。而使用 EMSCRIPTEN_KEEPALIVE 宏参数便可以解决此问题。

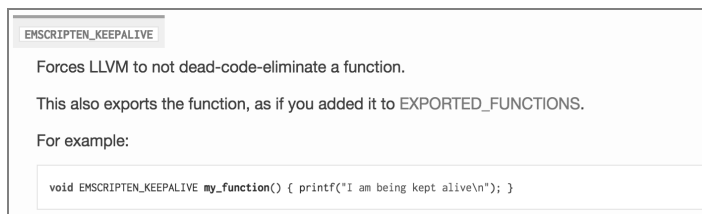


图5-14 Emscripten官方文档对“EMSCRIPTEN_KEEPALIVE”宏参数的用法详细描述

如图 5-15 所示，我们可以在 `emscripten.h` 文件中找到 EMSCRIPTEN_KEEPALIVE 宏参数的具体定义细节。可以看到，实际上 Emscripten 直接将该宏参数定义成了“`__attribute__((used))`”这个编译器描述符。而被标识为“`__attribute__((used))`”的函数定义则会被编译器强制地保留在目标代码中，即使该函数没有被任何主函数内的代码引用过。



图5-15 EMSCRIPTEN_KEEPALIVE宏参数的具体定义细节

当模块对应的 C/C++ 源代码编写完成后，我们便可以通过 Emscripten 工具链来进行编译流

程了。下面我们将通过两种方式来构建这个 Standalone 类型的 Wasm 应用。

使用增强型优化器 (Optimizer)

为了能够生成 Standalone 类型的 Wasm 应用，我们需要将目标模块文件中用于连接 ERE 环境 (Emscripten 运行时环境) 的所有参数和代码全部移除。这里将使用 emcc 自带的增强型优化器来移除目标文件中的多余内容。需要注意的是，这种方式可能并不适用于功能较为复杂，或使用了 C++ 11 及以上版本语法特性的 Wasm 应用。对应的编译命令如下。

```
emcc emscripten-standalone.cc
-Os
-s WASM=1
-o emscripten-standalone.js
```

在上面的命令语句中，我们通过“-s”参数为 emcc 指定了“WASM=1”标识符，以使其将 WebAssembly 作为编译目标的文件类型。而“-Os”参数则是优化的关键所在，该参数会告知编译器以“第 4 等级”的优化策略来优化目标代码，进而删除其中没有被使用到并且与 Emscripten 运行时环境相关的所有信息。这样，我们便得到了一个完全简化的独立型 Wasm 模块。接下来，我们可以使用如下 JavaScript 脚本代码（这里给出了完整的 HTML 文件内容），在 Web 浏览器中初始化该模块并使用其中暴露出的 C/C++ 方法。

```
index-optimizer.html
<!DOCTYPE html>
<html>
<head>
  <title>Emscripten - Standalone WebAssembly Module</title>
</head>
<body>

<script>
// 远程加载目标 Wasm 模块
fetch('emscripten-standalone.wasm').then(
  response => response.arrayBuffer()
).then(bytes =>
  // 没有需要向模块中导入的内容
  WebAssembly.instantiate(bytes, {})
).then(result => {
  // 从 exports 对象中获取模块对外暴露出的 add 方法
  const exportFuncAdd = result.instance.exports['_add'];
  // 调用该方法
```

```

    console.log(exportFuncAdd(10, 20));
  });
</script>
</body>
</html>

```

需要注意的是，在 **Name Mangling** 特性不生效的情况下，Emscripten 会将从 C/C++ 源代码中标记导出的函数的函数名前加上一个 “_” 字符作为前缀。因此，当我们从 `exports` 对象中获取导出函数时，需要使用 “_add” 来代替 C/C++ 源代码中的原始函数名 “add” 作为函数体在该对象中的索引键值。

编译成动态库（Dynamic Library）

这种方式是通过在编译命令中添加 “`SIDE_MODULE=1`” 标识来让 Emscripten 将 C/C++ 源代码文件编译成一个 WebAssembly 动态链接库的。

关于动态链接的相关内容，在本书第 3 章中做了介绍，当时通过直接修改 WAT 文件代码的方式模拟了 Wasm 模块之间的动态链接过程。而 Emscripten 作为一个用于构建 Wasm 应用的强大工具链，它在内部提供了一个可用于快速创建共享库模块的编译器标识符。我们可以通过如下命令将 C/C++ 源代码编译成一个 Wasm 类型的共享库模块文件。

```

emcc emscripten-standalone.cc
-s WASM=1
-s SIDE_MODULE=1
-o emscripten-dynamic-standalone.wasm

```

当模块编译完成后，我们可以通过下面给出的 HTML 文件来加载、初始化并执行从模块中暴露出的方法。与第一种方式不同的是，这里在初始化该 Wasm 模块时，我们需要向其内部导入包含有模块初始化资源的 `env` 命名空间对象，在这个对象中我们为模块提供了 `Table` 对象结构及相关的初始化参数（可以类比传统共享库又引用了另外一个共享库）。

```

index-dynamic.html
<!DOCTYPE html>
<html>
<head>
  <title>Emscripten - Standalone WebAssembly Module</title>
</head>
<body>

<script>

```

```
// 从远程加载目标 Wasm 模块
fetch('emscripten-dynamic-standalone.wasm').then(
  response => response.arrayBuffer()
).then(bytes =>
  WebAssembly.instantiate(bytes, {
    // 向模块中导入用于初始化的 env 模块对象
    env: {
      memoryBase: 0,
      tableBase: 0,
      table: new WebAssembly.Table({ initial: 2, element: 'anyfunc' }),
      abort: function (msg) {
        console.error(msg);
      }
    }
  })
).then(result => {
  // 从 exports 对象中获取模块对外暴露出的 add 方法
  const exportFuncAdd = result.instance.exports['_add'];
  // 调用该方法
  let result = exportFuncAdd(10, 20);
  console.log(result);
});
</script>
</body>
</html>
```

Dependent

Dependent 类型的 Wasm 应用与 Standalone 类型有所不同,在该类型应用中一般都包含着大量与浏览器特定功能相关的方法调用。比如在对应 C/C++源代码中使用了 IO 标准库、OpenGL 等需要与宿主环境本身进行交互的相关技术。Emscripten 在将这些与特定功能相关的 C/C++源代码转译为 ASM.js 代码时,需要对部分源代码进行单独处理,以使其能够符合浏览器的特殊运行时环境。另外,由于 Wasm 模块本身无法直接与浏览器进行交互,因此,Emscripten 便需要通过某种具有类似“胶水”功能的 JavaScript 代码,来将 Wasm 模块与 Web 浏览器在功能交互和数据资源传输层面连接起来。

在 Standalone 类型的 Wasm 应用中,“胶水”代码的作用十分简单,除对模块进行远程加载、实例初始化和导出方法调用之外,基本不再需要实现任何其他功能。因此,对于这部分代码,我们可以自己编写来简单地实现。但是对于 Dependent 类型的 Wasm 应用,通过手写的方式来

实现“胶水”代码便没有那么简单了。这是由于在 **Dependent** 类型应用对应的 C/C++ 源代码中，大量地使用了依赖浏览器特定实现的宿主环境和 **API** 接口，而为了“连接”这部分功能与 **WebAssembly** 模块实例，便需要通过 **JavaScript** 脚本来屏蔽 **Wasm** 模块与浏览器本身的交互细节，而实现这部分功能的工作量是巨大的。

幸运的是，**Emscripten** 帮助我们自动完成了这部分工作。**Emscripten** 在构建 **Dependent** 类型的 **Wasm** 应用时，会自动帮助我们生成用于连接模块与浏览器环境的“胶水”脚本文件。在这个文件中，**Emscripten** 构建了一个专门用于辅助浏览器与 **Wasm** 模块进行连接的 **JavaScript** 运行时体系，即“**Emscripten** 运行时环境(ERE)”。比如，我们尝试通过 **Emscripten** 将下面这段 C/C++ 源代码编译成一个 **Dependent** 类型的 **Wasm** 应用。

```
#include <emscripten.h>
#include <iostream>

using namespace std;

#ifdef __cplusplus
extern "C" {
#endif

// 函数定义，利用 Emscripten 提供的宏防止函数被 DCE 处理掉
EMSCRIPTEN_KEEPALIVE void echo (int x) {
    // 这里使用了标准库中的 cout 对象
    cout << "The number you input is: " << x << endl;
}

#ifdef __cplusplus
}
#endif
```

在这段代码中，我们使用了 C++ 标准库中的 **cout** 对象来向控制台打印从上层 **JavaScript** 环境传入的一个整型数据。接下来，我们将编写一小段 **JavaScript** 代码来调用后续 **Wasm** 模块暴露出的函数，以及其他相关的主流程代码。如下面的代码所示，这里在一个独立的 **post-script.js** 脚本文件中，我们通过 **Module.ccall** 全局函数调用了模块暴露到 **JavaScript** 环境中的 **echo** 方法，并同时向该函数传入一个数值“10”作为参数。

```
post-script.js
// 向 Module 初始化完毕的钩子队列中加入待执行的内容
__ATPOSTRUN__.push(() => {
```

```
// 调用模块中暴露出的 echo 方法
Module.ccall('echo', null, ['number'], [10]);
// 也可以通过这种方式来调用
Module['asm']['_echo'](10);
});
```

由于 Emscripten 会自动生成用于连接模块与浏览器的 JavaScript 脚本文件，因此，这里我们并不需要考虑应该如何加载模块，以及如何为模块提供初始化所需要的数据信息。我们只需要编写模块初始化后需要执行的一系列主流程代码即可。Emscripten 自动生成的脚本文件不仅帮助我们完成了模块的加载与初始化工作，而且还帮助我们封装了诸如 `Module.ccall` 等一系列可以直接用于调用模块内部函数的 JavaScript 方法，即“胶水”方法。

除此之外，在这段代码中我们还使用了一个名为“`__ATPOSTRUN__`”的数组结构，放入该数组结构中的函数将会在模块和 Emscripten 运行时环境初始化完成后被依次执行。因此，我们也将“`__ATPOSTRUN__`”对应的数组结构称为 Emscripten 运行时环境内部的一个生命周期钩子（Hook）队列。如图 5-16 所示，Emscripten 运行时环境在其内部定义了多种类型的钩子队列结构，而放置在这些队列内的函数将会在整个 ERE 生命周期的特定阶段被执行。

emscripten/src/preamble.js Lines 1515 to 1519 in bf0bdc5	
1515	var __ATPRERUN__ = []; // functions called before the runtime is initialized
1516	var __ATINIT__ = []; // functions called during startup
1517	var __ATMAIN__ = []; // functions called when main() is to be run
1518	var __ATEXIT__ = []; // functions called during shutdown
1519	var __ATPOSTRUN__ = []; // functions called after the runtime has exited

图5-16 定义在Emscripten运行时环境中的生命周期钩子队列

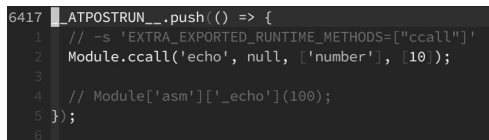
接下来，我们便可以通过如下命令来编译上面这段 C/C++ 源代码。

```
emcc dependent.cc
-s WASM=1
-s 'EXTRA_EXPORTED_RUNTIME_METHODS=["ccall"]'
--post-js post-script.js
-o dependent.js
```

这里我们为 `emcc` 命令指定了几个必要的参数。其中，“`WASM=1`”标识用于设置编译器生成的目标文件类型为 WebAssembly 二进制模块。“`EXTRA_EXPORTED_RUNTIME_METHODS`”标识以数组的形式记录了所有需要被导出的 Emscripten 运行时方法。由于我们在上述 JavaScript 脚本中使用了 Emscripten 运行时环境提供的 `ccall` 方法，因此，这里则需要将其显式地标记出来，以便 Emscripten 能够将该方法的定义直接绑定到全局的 `Module` 对象中。“`--post-js`”参数则用于指定需要被追加到“胶水”脚本文件的 JavaScript 代码，这里直接将之前创建的包含着主流程

代码的 `post-script.js` 脚本文件内容拼接到“胶水”脚本文件的尾部。

当上面的命令执行完毕后，我们可以在当前目录下找到由 Emscripten 工具链生成的“胶水”脚本文件。如图 5-17 所示，打开该文件并将光标移动至内容最后，可以看到我们之前写入 `post-script.js` 文件中的代码已经被追加到了这里。同理，使用“`--pre-js`”编译参数可以添加需要在 Module 对象初始化前执行的 JavaScript 脚本代码，即追加到脚本文件的头部。



```
6417 _ATPOSTRUN__push(() => {
1 // -s 'EXTRA_EXPORTED_RUNTIME_METHODS=["ccall']'
2 Module.ccall('echo', null, ['number'], [10]);
3
4 // Module['asm']['_echo'](100);
5 });
6
```

图5-17 被追加到“胶水”脚本文件中的`post-script.js`文件内容

最后，我们将通过如下 HTML 文件来整合 Wasm 模块，以及 Emscripten 生成的“胶水”脚本文件。与 Standalone 类型 Wasm 应用的初始化过程不同，这里需要在加载脚本文件之前，先使用从远程位置获取的 Wasm 模块二进制数据来填充 Module 全局对象的 `wasmBinary` 属性，然后再通过动态加载的方式将“胶水”脚本文件绑定并加载到当前的 HTML 文件中。“胶水”脚本文件中的代码在执行时，会自动检测在当前全局作用域内是否存在名为“Module”的 JavaScript 对象，并且该对象上的 `wasmBinary` 属性中是否包含一段有效的 Wasm 模块二进制数据。若一切正常，则脚本文件会自动完成 Emscripten 运行时环境初始化、模块的加载和实例化等过程。最后当 Wasm 模块初始化完毕后，位于 `post-script.js` 文件中的主流程代码将会被执行。

```
index.html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title></title>
</head>
<body>

<script>
  // 初始化 Module 全局对象，由 Emscripten 工具链自动完成内容填充
  var Module = {};

  fetch('dependent.wasm').then(
    response => response.arrayBuffer()
  ).then((bytes) => {
    // 填充模块数据
```



```
Module.wasmBinary = bytes;
// 动态异步地载入由 Emscripten 生成的“胶水”脚本文件
var script = document.createElement('script');
script.src = "dependent.js";
document.body.appendChild(script);
});
</script>
</body>
</html>
```

5.2.2 Emscripten 运行时环境

在上一节的 Wasm 应用中，我们使用到了 `Module.ccall` 等由 Emscripten 运行时环境为我们封装好的内部方法。这些方法在实现细节上屏蔽了 JavaScript 与 C/C++ 代码之间的运行时差异，这使得开发者可以更加专注于业务代码的实现，而不用过多考虑代码在两个运行时之间的转换问题。在本节中，将针对一些常见的编程语言差异问题，介绍 Emscripten 运行时环境的处理方法。

输入/输出

Emscripten 工具链为 Web 浏览器环境实现了一套与 SDL（Simple DirectMedia Layer，简单直接媒体层）库标准完全对应的 API 接口，通过这些接口我们可以将原先基于 SDL 开发的应用程序无痛地移植到 Web 浏览器上运行（在无须修改或只进行少量代码修改的情况下）。SDL 是一套跨平台的软件开发库，通过它我们可以直接访问计算机底层的音频、键盘、鼠标、游戏摇杆和图形硬件等输入/输出设备，进而它可以被用于开发高性能的视频游戏、游戏开发引擎，以及各类多媒体应用程序。SDL 库本身是基于 C 语言编写的，它为多种语言提供了相应的接口实现，如 C++、C#、Lua 和 Go 等。SDL 定义了多种上层用户与计算机底层物理输入/输出设备的交互方式，并将不同平台上的实现差异全部封装到统一的抽象层中，这使得开发者可以更加方便地编写跨平台的应用程序，SDL 也因此得到了广泛的应用。例如，大家所熟知的跨平台游戏“Angry Bird（愤怒的小鸟）”便是基于 SDL 库进行开发的。

除了使用 SDL，我们也可以选择使用 Emscripten 封装好的一系列特定接口，在 C/C++ 代码中与浏览器 HTML 5 本身的事件系统进行交互。Emscripten 在 `emsdk` 开发工具包中提供了一个 `html5.h` 头文件，在该头文件中定义了一系列可用于在 C/C++ 源代码中处理键盘事件、鼠标事件、设备方位、屏幕触摸和页面可见性等与底层 I/O 设备相关的交互逻辑接口。这些接口被紧密地映射到与其等效的 JavaScript 接口上。除此之外，Emscripten 还对如下几种常用的底层图形设备处理库提供了对等的 Web 浏览器支持方案。

- **GLUT**: GLUT (OpenGL Utility Toolkit) 是一个用于开发 OpenGL 应用程序的工具库, 它为 OpenGL 实现了一套简单的窗口化 API, 进而使得学习和探索 OpenGL 编程变得相当容易。通过 GLUT, 我们能够在不了解 X Window、Microsoft Windows 窗口系统实现原理的情况下, 简单地编写出跨平台的 3D 应用程序。
- **GLFW**: GLFW (Graphics Library Framework) 是一个专门用于编写 OpenGL 应用程序的轻量级 C 语言库, 它同样为开发者提供了用于创建窗口、管理 OpenGL 上下文和相关 I/O 物理设备的能力。从整体上看, GLFW 的开发目的主要是用于替代上述古老的 GLUT 工具库, 它同样支持构建跨平台的 OpenGL 应用程序。
- **GLEW**: GLEW (OpenGL Extension Wrangler Library) 是一个跨平台且开源的 C/C++ 扩展加载库。GLEW 提供了高效的运行时机制, 用于检测目标平台支持哪些版本和类型的 OpenGL 扩展接口。GLEW 从整体上屏蔽了各种型号显卡所支持的不同接口类型, 它把整个 OpenGL 的核心及扩展功能全部暴露在单个头文件中。
- **XLib**: XLib (The X Library) 是一个基于 C 语言编写的实现了 X Window 和 X Server 交互层协议的底层函数库, 通过它我们可以在不知道相关协议细节的情况下编写桌面应用程序。但是由于其 API 较为底层, 我们很少直接使用它来构建大型的桌面客户端应用程序。

除上述介绍的计算机物理设备的输入/输出以外, 对于诸如基于 `cin/cout` 实现的控制台输入/输出, Emscripten 会使用 Web 浏览器提供的 `window.console` 对象与 `window.prompt` 方法来模拟对应的 I/O 过程。

文件系统

传统的 C/C++ 应用一般会使用 `libc` 和 `libcxx` 标准库中提供的同步文件系统 API 来访问本地文件, 但由于浏览器的安全策略限制, 我们无法直接在浏览器环境中访问本地主机系统中的文件。不仅如此, 曾经提出的用于在 Web 浏览器中模拟本地文件系统的“File and Directory Entries API”标准也仅有 Chrome 浏览器支持。为此, Emscripten 专门构建了一套针对 `libc` 和 `libcxx` 库的浏览器实现, 同时还构建了一个浏览器端的虚拟文件系统, 以便能够正常编译并在 Web 浏览器上运行使用了标准库文件操作相关 API 的 C/C++ 程序, 而不用进行任何修改。

相比于传统的本地文件系统, Emscripten 所构建的虚拟文件系统则需要在应用运行前进行初始化。所谓的初始化是指将应用需要在本地文件系统中使用的所有文件, 事先全部通过异步的 JavaScript API 加载到浏览器的虚拟文件系统中。随后, 应用在运行过程中进行的所有文件读取操作, 实际上将会间接使用这些提前被缓存到浏览器端的文件资源。

该虚拟文件系统具有多种文件系统类型可供选择，其中默认的文件系统 MEMFS 将文件存储在浏览器内存中，因此任何页面的重新加载都会使这些文件内容丢失；文件系统 IDBFS 则允许将文件数据长时间地保留在浏览器中以进行持久化处理；而在 Node.js 中运行代码时，我们可以选择通过文件系统 NODEFS 来直接访问计算机本地的真实文件系统。除此之外，Emscripten 还提供了支持异步文件访问的 API，这部分内容我们将在后续的章节中进行介绍。

浏览器主循环

Web 浏览器内的事件模型使用了“协同多任务”处理机制来控制每一个事件的实际发生过程。即每一个事件每次只能在一个等长的时间片内运行，在该时间片结束后必须将控制权返回给浏览器，以便其他事件继续执行。通常，导致 HTML 页面无响应的一个常见原因就是对应于当前事件的 JavaScript 脚本没有及时退出并将控制权返回给浏览器，比如死循环或函数的无限递归调用。而传统的 C/C++ 图形应用通常会以无限循环的方式来执行同样的一段代码（通常为渲染函数，比如 GLUT 中的 `glutMainLoop` 函数），在这段代码中，应用程序分别负责处理事件、图形的计算和渲染，整个过程会通过添加延迟的方式来确保每一帧所花费的时间保持一致。但这种无限循环的代码执行方式无法直接在浏览器环境中使用，因为控制权无法在时间片内返回给浏览器，浏览器将会通知用户页面无响应并停止提供服务。

同样的，在浏览器环境中像 WebGL 这样的 JavaScript API 也只能在浏览器的时间片结束时被执行，并且在这些 API 内部会自动渲染和交换当前视图缓冲区中的内容。而这与基于 OpenGL 构建的 C/C++ 应用程序则正好相反，在这里需要手动交换缓冲区的内容，以实现屏幕视图中显示内容的更新。Emscripten 对这个问题的处理方案则是专门定义一个 C/C++ 函数来代替图像处理代码中的无限循环过程，如下所示为来自官方文档中的示例代码。

```
int main (int argc, char **argv) {
    ...
    // 利用 “__EMSCRIPTEN__” 宏来区分编译环境
    #ifdef __EMSCRIPTEN__
        // void emscripten_set_main_loop(em_callback_func func, int fps, int
        simulate_infinite_loop);
        emscripten_set_main_loop(one_iter_render, 60, 1);
    #else
        while (1) {
            one_iter_render();
            // 延迟绘制，使帧率稳定
            SDL_Delay(time_to_next_frame());
        }
    #endif
}
```

```
// 用于无限循环/主循环的绘制函数
void one_iter_render() {
    // 处理数据
    // 绘制视图
}
```

在这段代码中，我们通过宏参数“`__EMSCRIPTEN__`”来区分当前代码的编译环境是否为 Emscripten 工具链内部（`emcc`）。而在 Emscripten 所对应的编译代码段中，我们会使用主循环函数 `emscripten_set_main_loop` 来让运行时环境以指定的频率调用传递给该函数的绘制函数。比如在上面的代码中，我们将用于屏幕视图绘制的绘制函数 `one_iter` 作为参数传递给该函数。在实际运行时，Emscripten 运行时环境会以固定每秒 60 次的速度来调用该绘制函数以更新页面的视图内容。而这不同于无限循环的是，其他浏览器事件对应的 JavaScript 代码可以在每次主循环函数调用之间执行，浏览器的控制权可以在绘制函数与其他事件之间共享，因此不会出现无响应的问题。

执行生命周期

每一个基于 Emscripten 工具链构建的 Wasm 应用都会经历从 Emscripten 运行时环境创建到应用业务代码执行，再到应用退出及运行时环境销毁的过程，而在这中间则对应着多个不同的生命周期节点。在每一个生命周期的关键节点上，Emscripten 都提供了相应的钩子函数来方便开发者实现诸如资源预处理、节点日志记录等多种形式的业务需求。如图 5-18 所示为一个基于 Emscripten 工具链构建的 Wasm 应用其生命周期中各关键节点的基本执行顺序。

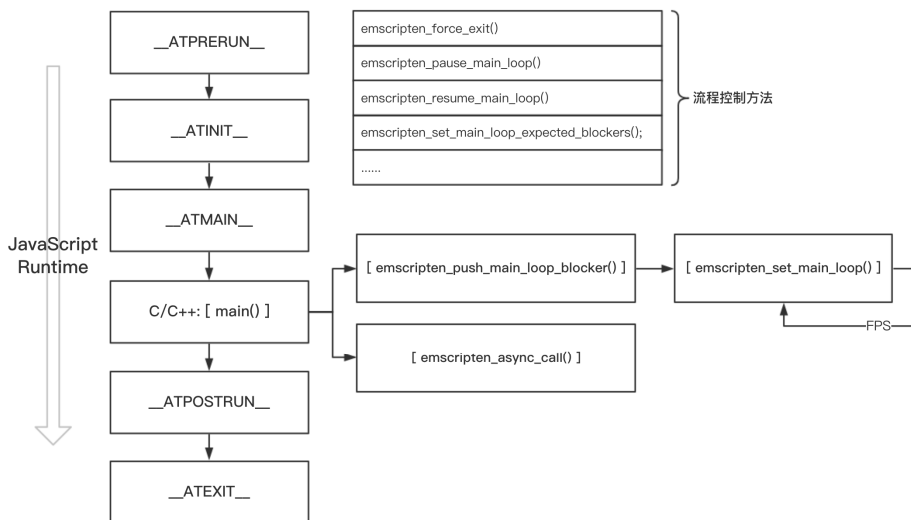


图5-18 基于Emscripten工具链构建的Wasm应用其生命周期中各关键节点的基本执行流程图

在图 5-18 中, 最左侧一列以 “__” 开头和结尾的标识符为 Emscripten 在 JavaScript 运行环境中提供的钩子队列数组。比如 “__ATPOSTRUN__” 队列, 在整个 Wasm 应用的生命周期中, 当对应 C/C++源代码中主函数的代码逻辑执行完毕后, 被放入该队列数组内的函数会开始执行; “__ATPRERUN__” 队列中的函数, 会在应用开始运行前被执行, 这里一般会进行虚拟文件系统的初始化工作; “__ATINIT__” 队列中的函数, 会在 Emscripten 运行时环境开始初始化时被执行; “__ATMAIN__” 队列中的函数, 会在 C/C++源代码中的主函数将被调用前被执行; “__ATEXIT__” 队列中的函数, 会在整个应用/ERE 运行时环境退出时被执行, 在这里可以对之前分配的系统 and 内存资源进行回收处理。

上面介绍的 5 种钩子队列可以在应用对应的 JavaScript 代码中直接使用。相应的, Emscripten 也提供了一些可以应用在 C/C++源代码中的生命周期函数, 这些生命周期函数主要用于辅助模拟浏览器的主循环流程, 并对该流程进行控制。下面对这些生命周期函数的功能进行简单介绍。

`emscripten_set_main_loop()`

该函数是用于在 C/C++代码中模拟浏览器主循环的主要函数。从上一节的示例中可以看到, 该函数会以一个固定的帧速率重复不断地调用传递给它的绘制函数。实际上, Emscripten 在编译该函数时, 会将其直接转换为 JavaScript 环境下对应的 `setTimeout` 和 `requestAnimationFrame` 函数来模拟每秒固定次数的代码调用过程。

`emscripten_push_main_loop_blocker()`

该函数用于在 `emscripten_set_main_loop` 函数运行前为其添加相应的预处理过程。比如在构建一款游戏时, `emscripten_set_main_loop` 函数主要用于对游戏的显示画面进行实时的渲染控制, 同时负责推进游戏的发展进度。一般来说, 在游戏进入主流程之前, 都需要预先加载诸如贴图、音/视频等多媒体资源, 而这个流程便可以放置到 `emscripten_push_main_loop_blocker` 函数中进行。该函数会阻塞主循环函数的执行流程, 直至该函数被完全执行完毕。不仅如此, 我们还可以通过 JavaScript 运行时环境提供的 `Module.setStatus` 回调函数, 来实时地检测位于执行队列中的 `emscripten_push_main_loop_blocker` 函数其总体完成进度, 并将结果同步地反馈给用户。

`emscripten_async_call()`

该函数主要用来在 C/C++源代码中异步执行一段代码。与 `emscripten_set_main_loop` 函数不同的是, 该函数不会被任何方法阻塞, 因此我们可以用它来执行任何需要在特定时间段后立即运行的任务。

除上面介绍的几个用于模拟特定代码执行流程的函数以外, 还有几个可以被应用在这些函数

上，用于控制函数整体执行流程状态的函数，如下所示。

`emscripten_pause_main_loop()`

该函数主要用于暂停当前浏览器主循环的执行流程。

`emscripten_resume_main_loop()`

该函数主要用于恢复当前浏览器主循环的执行流程，它通常与上一个函数配套使用。

`emscripten_force_exit()`

该函数用于停止并强制退出当前的 Wasm 应用。函数被调用后会直接触发当前 JavaScript 运行时环境中的 “`__ATEXIT__`” 钩子队列。若要在 C/C++ 源代码中使用该函数，则需要在编译时为 `emcc` 指定 “`NO_EXIT_RUNTIME=0`” 参数，以允许 Emscripten 运行时环境的退出。

`emscripten_set_main_loop_expected_blockers()`

该函数主要用于向 Emscripten 运行时环境提前报告将要在 `emscripten_set_main_loop` 主循环函数执行前执行的预处理函数 (`emscripten_push_main_loop_blocker`) 个数。这样做的好处在于，Emscripten 可以根据当前已经完成的预处理函数个数，通过 `Module.setStatus` 回调函数告知用户当前的数据预处理进度，其应用场景如游戏的关卡加载进度画面。

接下来，我们将尝试使用上面介绍的生命周期函数和流程创建/控制函数。通过在浏览器中直观地打印出每一个函数的实际调用时机，相信读者会对 Emscripten 下的 WebAssembly 应用其生命周期系统有更加深刻的理解。首先给出的是该 Wasm 应用对应的 C/C++ 源代码。

```
lifecycle.cc
```

```
#include <emscripten.h>
#include <iostream>
#include <string>

using namespace std;
// 定义必要的控制变量
int counter = 0;
int blockerNumbers = 5;
int *p = &counter;
// 定义主循环函数
void one_iter_render (void) {
    cout << "[emscripten_set_main_loop] Prints from main loop ..." << counter++ << endl;
```

```

// 当 counter 变量的值为“5”时执行
if (counter == 5) {
    // 停止主循环
    emscripten_pause_main_loop();
    // 强制终止并退出应用
    emscripten_force_exit(1);
}
}
// 在主循环之前执行的预处理函数
void one_iter_block (void* args) {
    cout << "[emscripten_push_main_loop_blocker] Prints from main loop blocker ..."
        << *static_cast<int*>(args) << endl;
}
// 需要异步执行的代码逻辑
void one_iter_async (void* args) {
    cout << "[emscripten_async_call] Prints from JavaScript context async ..." << endl;
}
// C/C++主函数
int main (int argc, char **argv) {
    // 设置 5 个预处理函数（会阻塞主循环函数的执行）
    emscripten_push_main_loop_blocker(one_iter_block, p);
    emscripten_push_main_loop_blocker(one_iter_block, p);
    emscripten_push_main_loop_blocker(one_iter_block, p);
    emscripten_push_main_loop_blocker(one_iter_block, p);
    emscripten_push_main_loop_blocker(one_iter_block, p);

    // 向 Emscripten 报告预处理函数的个数
    emscripten_set_main_loop_expected_blockers(blockerNumbers);
    // 设置异步代码的执行条件（2000ms 后执行）
    emscripten_async_call(one_iter_async, nullptr, 2000);
    // 设置主循环函数
    emscripten_set_main_loop(one_iter_render, 1, 1);

    return 0;
}

```

为了能够使用定义在 JavaScript 运行时环境中的钩子队列，我们还需要编写如下 JavaScript 脚本代码。在这段脚本代码中，我们分别在 5 个钩子队列中放置了需要执行的测试函数，每一个函数都会在执行时向浏览器打印出当前函数所在钩子队列的相关信息。整个脚本代码将会通过 emcc 编译器的“--post-script”参数被追加到“胶水”脚本文件的尾部来执行。

```

post-script.js
// 向各 Emscripten 生命周期钩子队列中添加待执行的匿名函数
__ATPRERUN__.push(() => {
  console.log("__ATPRERUN__ Prints from pre-loading stage ...");
});

__ATPOSTRUN__.push(() => {
  console.log("__ATPOSTRUN__ Prints from post-loading stage ...");
  // 设置 Module.setStatus 回调函数，即时打印出 C/C++ 源代码中预处理函数的当前执行状态
  Module.setStatus = function (status) {
    console.log("[Module.setStatus]", status);
  }
});

__ATINIT__.push(() => {
  console.log("__ATINIT__ Prints from init stage ...");
});

__ATMAIN__.push(() => {
  console.log("__ATMAIN__ Prints from main stage ...");
});

__ATEXIT__.push(() => {
  console.log("__ATEXIT__ Prints from exit stage ...");
});

```

当一切准备就绪后，我们可以通过如下命令使用 `emcc` 来编译并生成该 Wasm 应用。需要注意的是，在编译时需要为 `emcc` 指定 “`NO_EXIT_RUNTIME=0`” 参数，这样才能够在 C/C++ 源代码中通过 `emscripten_force_exit` 函数强制结束应用并完全退出 Emscripten 运行时环境。

```

emcc lifecycle.cc
-s WASM=1
-o lifecycle.html
--post-js post-script.js
-s NO_EXIT_RUNTIME=0

```

当编译完成后，Emscripten 将自动生成可用于浏览器的 HTML 文件。同样的，这里我们仍然需要一个静态的 HTTP 服务器来提供 HTTP 服务。由于 Emscripten 在 “胶水” 脚本文件中会优先尝试使用 `WebAssembly.instantiateStreaming` 方法来加载远程的 Wasm 二进制模块，因此该 HTTP 服务器需要在接收到相应的 “.wasm” 类型文件请求时，返回与其相对应的 MIME 编码类

型，即 `application/wasm` 类型。但由于现阶段大部分静态服务器的 MIME 映射表中并不默认支持该文件类型，因此我们需要通过 Node.js 来自己构建一个小型的 HTTP 静态服务器以提供服务。关于这部分内容可以参考本书前面的介绍，这里不再赘述。

在应用运行后，我们可以从浏览器的 Console（控制台）中得到如图 5-19 所示的输出信息。参考图 5-18，可以很容易地将各行输出语句与源代码中的各个节点函数相对应。

在图 5-19 中，每行输出内容的前半部分都通过 “[]”（方括号）的形式标记出了对应该行内容的打印函数名，可以看到 `Module.setStatus` 函数会在每次 `emscripten_push_main_loop_blocker` 预处理函数的任务执行完毕后，向用户侧通知应用当前预处理过程的整体进度。当所有预处理函数对应的任务都被执行完毕后，主循环函数 `emscripten_set_main_loop` 才会开始执行。而异步的事件调用函数 `emscripten_async_call` 则不受任何影响。最后，当通过调用 `emscripten_force_exit` 函数来强制终止应用并退出当前的 Emscripten 运行时环境时，在 JavaScript 代码中被放置在 “`__ATEXIT__`” 钩子队列中的函数将会被执行。

10:49:32.611	Fetch finished loading: GET "http://localhost:8888/lifecycle.wasm".	lifecycle.js:1601
10:49:32.886	[__ATPRERUN__] Prints from pre-loading stage ...	lifecycle.js:7250
10:49:32.892	[__ATINIT__] Prints from init stage ...	lifecycle.js:7261
10:49:32.893	[__ATMAIN__] Prints from main stage ...	lifecycle.js:7265
10:49:32.896	[__ATPOSTRUN__] Prints from post-loading stage ...	lifecycle.js:7254
10:49:32.899	[Module.setStatus]	lifecycle.js:7256
10:49:32.905	[emscripten_push_main_loop_blocker] Prints from main loop blocker ...0	lifecycle.html:1237
10:49:32.909	main loop blocker "one_iter_block" took 7 ms	lifecycle.js:5193
10:49:32.910	[Module.setStatus] Please wait... (1/5)	lifecycle.js:7256
10:49:32.912	[emscripten_push_main_loop_blocker] Prints from main loop blocker ...0	lifecycle.html:1237
10:49:32.914	main loop blocker "one_iter_block" took 2 ms	lifecycle.js:5193
10:49:32.915	[Module.setStatus] Please wait... (2/5)	lifecycle.js:7256
10:49:32.919	[emscripten_push_main_loop_blocker] Prints from main loop blocker ...0	lifecycle.html:1237
10:49:32.923	main loop blocker "one_iter_block" took 4 ms	lifecycle.js:5193
10:49:32.924	[Module.setStatus] Please wait... (3/5)	lifecycle.js:7256
10:49:32.931	[emscripten_push_main_loop_blocker] Prints from main loop blocker ...0	lifecycle.html:1237
10:49:32.937	main loop blocker "one_iter_block" took 6 ms	lifecycle.js:5193
10:49:32.938	[Module.setStatus] Please wait... (4/5)	lifecycle.js:7256
10:49:32.944	[emscripten_push_main_loop_blocker] Prints from main loop blocker ...0	lifecycle.html:1237
10:49:32.947	main loop blocker "one_iter_block" took 3 ms	lifecycle.js:5193
10:49:32.948	[Module.setStatus]	lifecycle.js:7256
10:49:32.957	[emscripten_set_main_loop] Prints from main loop ...0	lifecycle.html:1237
10:49:33.957	[emscripten_set_main_loop] Prints from main loop ...1	lifecycle.html:1237
10:49:34.898	[emscripten_async_call] Prints from JavaScript context async ...	lifecycle.html:1237
10:49:34.956	[emscripten_set_main_loop] Prints from main loop ...2	lifecycle.html:1237
10:49:35.959	[emscripten_set_main_loop] Prints from main loop ...3	lifecycle.html:1237
10:49:36.960	[emscripten_set_main_loop] Prints from main loop ...4	lifecycle.html:1237
10:49:36.963	[__ATEXIT__] Prints from exit stage ...	lifecycle.js:7269

图5-19 上述Wasm应用在浏览器中的实际运行结果

限于篇幅，这里我们只介绍了 Emscripten 中常用的生命周期函数，其实还有很多与浏览器运行环境相关的函数，可以在工具链内的 `emsdk/emscripten/<version>/system/include/emscripten/emscripten.h` 文件中找到它们。读者可以参考 Emscripten 的官方文档来进一步了解这些函数的具体使用方法。

Emscripten 内存表示

Emscripten 使用单一的“typed array（类型数组）”在 JavaScript 环境与 Wasm 模块之间传递数据。类型数组本身是基于 `ArrayBuffer` 构建的一段连续的二进制缓存区。与普通 JavaScript 数组不同的是，通过类型数组来存储数据可以以最小独立单元“比特”为单位（同 C/C++），这在某种程度上可以节省数据存储所占用的内存空间。另外，基于类型数组，我们可以通过不同的类型视图来加载和读取内存中的数据。视图的类型决定了存储或读取数据时的数据类型，比如 `HEAPU32` 表示以 32 位即 4 个字节为整体，从类型数组中读取或写入一个整数。

5.2.3 在 JavaScript 代码中调用 C/C++ 函数

在构建 WebAssembly 应用时，一个最常见的需求就是希望能够在 JavaScript 代码中调用从 C/C++ 源代码中暴露出的函数。本节我们将讨论在整个由 Emscripten 构建的 Wasm 应用其最上层的 JavaScript 代码中，应该如何调用从 Wasm 模块中导出的 C/C++ 函数。

借助 Emscripten 运行时环境，我们可以直接通过其内部已经封装好的 `ccall` 和 `cwrap` 两个方法来方便地调用模块中的 C/C++ 函数。

ccall

`ccall` 方法主要用于直接调用从 Wasm 模块中导出的函数。下面我们将通过一个简单的示例应用来了解该方法的实际用法。首先给出的是该 Wasm 应用对应的 C/C++ 代码。

```
ccall.cc
#include <cmath>
#include <emscripten.h>

extern "C" {
    double EMSCRIPTEN_KEEPALIVE doubleSqrt(double x) {
        return sqrt(x);
    }
}
```

在这段代码中，我们定义了一个 `doubleSqrt` 函数，该函数用于计算所给定输入参数的平方根值。这里将该函数的整体定义放置在 `extern` 语句中，以防止 C++ 的 Name Mangling 机制导致

从模块中导出的函数的函数名发生变化。在函数定义中，宏参数 `EMSCRIPTEN_KEEPALIVE` 用于告知编译器在进行代码 DCE 优化时保留该函数的完整定义。

接下来，我们将编写当模块加载完毕后需要执行的 JavaScript 主业务逻辑代码。在这部分代码中，将主要通过 Emscripten 运行时环境提供的 `ccall` 方法来调用从模块中导出的 `doubleSqrt` 函数。这里将主业务逻辑代码以匿名函数的形式放置到 Emscripten 运行时环境的“`__ATPOSTRUN__`”钩子队列中。当模块加载完成后将自动执行这部分代码。

```
post-script.js
__ATPOSTRUN__.push(() => {
  // 调用 Module 全局对象上的 ccall 方法
  const result = Module.ccall('doubleSqrt', 'number', ['number'], [400]);
  // 打印函数调用结果
  console.log(result);
});
```

在上面的代码中，可以看到 `ccall` 方法的具体使用细节。该方法一共接收 4 个主要参数，对它们的说明分别如下：

- 第一个参数：待调用的从 Wasm 模块中导出的函数的函数名。这里直接传入字符串形式的 C/C++ 函数名“`doubleSqrt`”即可。
- 第二个参数：用于指定该 C/C++ 函数的具体返回值类型。这里在 Emscripten 运行时环境中可以使用的返回值类型有 4 种，分别是用于表示所有数字值（整数+浮点数）的“`number`”类型；用于表示字符串的“`string`”类型；用于表示布尔值的“`boolean`”类型；用于表示数组结构的“`array`”类型。由于被定义在 C/C++ 源代码中的 `doubleSqrt` 函数，需要在经过计算后返回一个双精度浮点数结果值，因此我们需要在 `ccall` 方法中指定该函数的返回值类型为“`number`”。
- 第三个参数：用于指定待调用 C/C++ 函数可接收的参数值类型。实际上，函数可接收的参数个数并不确定，因此需要将类型字符串按照参数的对应位置全部放置到统一的 JavaScript 数组中。这里由于 `doubleSqrt` 函数最多只能接收一个参数值，因此我们直接将参数类型字符串“`number`”放置到一个字面量数组中并传递给 `ccall` 方法。
- 第四个参数：用于指定需要向函数传递的实际参数值。我们将所有希望传递给该函数的参数按照其形参的对应排列顺序依次放置到统一的 JavaScript 数组中。这里将唯一的数字参数值“`400`”放置到一个字面量数组中并传递给 `ccall` 方法。注意，这里传入的实参类型需要与在上一个参数中指定的对应参数值类型相符合。

至此，我们便完成了对 `ccall` 方法的调用过程。如图 5-20 所示为在 Emscripten “胶水”脚本文件中 `ccall` 方法的代码实现细节。可以看到，该方法在实际执行时，首先会通过第一个名为“`ident`”的参数来查找从 Wasm 模块中导出的 C/C++ 函数实体，而真实的导出函数名则是由“`_`”作为前缀组成的，这与我们之前在构建 Standalone 类型 Wasm 应用时所使用的导出函数名特征相一致。接下来，该方法会根据所指定的形参类型将传入导出函数的实参转换成可用于该函数的数据类型。当参数转换完毕后，该方法便直接调用了模块的导出函数。对于最后得到的函数运算结果，也需要通过所指定的函数返回值类型进行相应的 JavaScript 数据类型转换。比如对于布尔类型的结果，这里会将从 C/C++ 函数中获得的“0”和“1”值分别转换为 JavaScript 语言中的“`true`”和“`false`”；而对于字符串类型的函数返回结果，则需要通过另外的 `Pointer_stringify` 内部方法从 Wasm 模块的共享线性内存中提取相应的数据。

```
// C calling interface.
function ccall (ident, returnType, argTypes, args, opts) {
  // "_" + ident;
  var func = getCFunc(ident);
  var cArgs = [];
  var stack = 0;
  assert(returnType !== 'array', 'Return type should not be "array."');
  if (args) {
    for (var i = 0; i < args.length; i++) {
      // 查询类型转换函数;
      var converter = toC[argTypes[i]];
      if (converter) {
        if (stack === 0) stack = stackSave();
        // 转换类型;
        cArgs[i] = converter(args[i]);
      } else {
        cArgs[i] = args[i];
      }
    }
  }
  // 调用 Wasm 模块中的 C/C++ 方法;
  var ret = func.apply(null, cArgs);
  // 将返回值转换为对应的 JavaScript 类型;
  if (returnType === 'string') ret = Pointer_stringify(ret);
  else if (returnType === 'boolean') ret = Boolean(ret);
  if (stack !== 0) {
    stackRestore(stack);
  }
  return ret;
}
```

图5-20 `ccall`方法的JavaScript代码实现细节

对于上述 Wasm 示例应用，可以通过如下命令进行编译。待编译完成后，便可通过我们之前基于 Node.js 编写的小型 HTTP 服务器在 Web 浏览器中加载和运行该应用。

```
emcc ccall.cc
-o ccall.html
-s WASM=1
-s EXTRA_EXPORTED_RUNTIME_METHODS='["ccall"]'
--post-js post-script.js
```

接下来，我们将继续改进这个应用，以使它能够在 C/C++ 以及 JavaScript 环境之间进行诸如字符串、数组等复杂类型数据的交互过程。修改后的 C/C++ 源代码如下。

```
ccall.cc
#include <cmath>
#include <emscripten.h>

extern "C" {
    double EMSCRIPTEN_KEEPALIVE doubleSqrt(double x) {
        return sqrt(x);
    }

    // 参数为 C-style 字符串
    unsigned char* EMSCRIPTEN_KEEPALIVE capitalize (unsigned char *string) {
        int i = 0;
        while (1) {
            // 获取每个位置字符的 ASCII 编码值
            char _current_pos = *(string + i);

            // 转换为大写形式
            if (_current_pos >= 0x61 && _current_pos <= 0x7a) {
                *(string + i) = _current_pos - 32;
            }

            // 如果处理到字符串结尾，则退出循环
            if (_current_pos == '\0') {
                break;
            }

            i++;
        }
        return string;
    }

    // 将数组内所有元素的值增加 1
    char* EMSCRIPTEN_KEEPALIVE increment (char array[], int length) {
        for (int i = 0; i < length; i++) {
            array[i] = 1;
        }
    }
}
```

```

    }
    return array;
  }
}

```

在上面的代码中，可以看到我们增加了 C/C++ 字符串和数组两种复杂数据类型在 Wasm 模块与 JavaScript 环境之间的交互流程。其中 `capitalize` 函数会将从上层 JavaScript 环境传入的英文字符串转换为对应的大写形式；`increment` 函数则会将从上层 JavaScript 环境传入的一个数组对象其内部的所有元素值均增加 1。关于这部分代码功能的 C/C++ 实现细节，由于其逻辑较为简单，读者可以参考注释自行理解。这里我们将重点关注 Emscripten 运行时环境对字符串和数组这类复杂数据类型的值处理方式。现在我们通过如下 JavaScript 脚本代码来加载和使用模块中定义的这几个函数。同样的，这段代码整体上需要以匿名函数的形式进行封装，然后放置到“`__ATPOSTRUN__`”钩子队列中来执行。

```

post-script.js
__ATPOSTRUN__.push(() => {
  // "doubleSqrt"
  ...

  // "capitalize"
  const capitalizeResult = Module.ccall('capitalize', 'string', ['string'], ["yhspy"]);
  console.log(capitalizeResult);

  // "increment"
  let array = [1, 2, 3, 4];
  // 返回数组首地址
  const arrayPointer = Module.ccall('increment', 'number', ['array', 'number'], [array,
array.length]);
  // 定义一个结果集容器
  let clearArrResult = [];
  for (let i = 0; i < array.length; i++) {
    // 通过 Emscripten 运行时环境的 Module.getValue 函数从模块共享线性内存中提取数据
    clearArrResult.push(Module.getValue(arrayPointer + i, 'i8'));
  }
  console.log(clearArrResult);
});

```

接下来，我们通过如下命令来编译该 Wasm 应用。

```

emcc ccall.cc
--std=c++11

```

```
-o ccall.html
-s WASM=1
-s EXTRA_EXPORTED_RUNTIME_METHODS=['ccall', 'getValue']
--post-js post-script.js
```

我们先从 `capitalize` 函数的调用过程开始看起，这里直接通过 `ccall` 方法调用该函数，并且设置函数的形参类型与返回值类型均为 `string`。最后向该函数传入了一个值为“yhspy”的字符串字面量来作为函数调用的实际参数。通过观察 `ccall` 方法的具体实现代码，可以看到 Emscripten “胶水”脚本在处理字符串类型参数时的细节信息。首先，`toC` 函数会根据当前的参数类型，选择具体的数据处理函数对参数进行类型处理。在这里字符串类型的参数会直接对应于 `stringToC` 处理函数，该函数的代码实现细节如图 5-21 所示。

```
'stringToC' : function(str) {
  var ret = 0;
  if (str !== null && str !== undefined && str !== 0) { // null string
    // at most 4 bytes per UTF-8 code point, +1 for the trailing '\0'
    // 每一个字符占用4比特内存空间(所有Unicode字符);
    var len = (str.length << 2) + 1;
    // 分配内存并返回起始地址;
    ret = stackAlloc(len);
    // 填充内容, 各字符所对应的UTF8编码数据值;
    stringToUTF8(str, ret, len);
  }
  // 返回该段内存首地址;
  return ret;
};
```

图5-21 stringToC函数的代码实现细节

实际上，由于在 C/C++ 语言中字符串类型并不是一种基本数据类型，因此我们无法直接将 JavaScript 中的字符串以基本数据的形式传递给 C/C++ 代码进行处理。但是在 C/C++ 中却可以通过“字符串指针”来引用位于内存中的一段连续的字符值，并且只要该字符值最后能够以“\0”字符结尾，那么就可以通过字符串指针来操作位于这段连续内存中的所有字符值。这便是 C/C++ 语言中字符串类型（非标准库）的基本表示形式。为此，这里需要使用到 Wasm 模块的共享线性内存将 JavaScript 环境下的字符串值分解为单个的字符值，并将这些值存放到模块共享线性内存的一段连续空间中。这样，我们便可以在 C/C++ 代码中以字符串指针的形式从共享内存段中获取到来自 JavaScript 环境的字符串值内容。

从图 5-21 所示的代码中可以看到，Emscripten “胶水”脚本对 JavaScript 字符串类型参数的数据处理过程分为 3 个步骤。

计算占用的最大内存空间大小

在这一步中，Emscripten 计算需要为字符串数据申请的最大内存空间大小。一般来说，由

于 UTF-8 编码的可变长特性，我们无法在接触到具体字符串内容前就计算出将要使用的内存空间大小。因此，Emscripten 将按照每一个字符最多占用 4 字节内存（通过字符串长度左移 2 位）来计算所需要申请的内存空间大小。而字符串最后的“\0”也同样会占用 1 个字节的内存空间。

分配内存空间

在这一步中，Emscripten 使用名为“stackAlloc”的内部函数来完成共享线性内存段空间的分配过程。该函数会返回所分配的内存段的首地址，该地址将以一个 JavaScript 整数值的形式来表示。

填充数据

当内存空间分配完成后，就可以通过名为“stringToUTF8”的内部函数来向内存段中填充数据了。该函数在其内部通过原生的 charCodeAt 方法来获取字符串中每一个字符的 UTF-16 编码值。接下来，函数会按照从 UTF-16 到 UTF-8 编码的转换步骤来依次转换所得到的 UTF-16 编码值，转换后得到的 UTF-8 编码值将被直接存放到共享线性内存段的对应位置上。

从总体上看，对于字符串这种复杂的数据类型，Emscripten 通过 Wasm 模块的共享线性内存段以原始字节流的方式在 JavaScript 环境与 C/C++ 之间传递数据。同样的，在 C/C++ 代码中，对于字符串类型的返回值，则需要直接返回该字符串对应的字符串指针，以方便上层 JavaScript 环境直接从共享线性内存中来获取这些字符串对应的字符值。比如在 ccall 方法的实现代码中调用的 Pointer_stringify 内部方法，便主要用于通过字符串指针来解码对应的字符串内容。

接下来，我们继续观察模块中 increment 函数的具体使用方法。在 C/C++ 代码中，该函数将接收一个字符数组的首地址，然后将该首地址所对应数组内的所有元素值均增加“1”后又将该地址返回。事实上，通过 ccall 方法调用该函数的具体方式与前面的 capitalize 函数没有任何本质区别，只不过这里需要将函数的形参类型标记为数组（array），同时将其返回值类型标记为数字值（Number），该数字值用于表示指向共享线性内存段中某位置的指针。

Emscripten“胶水”脚本在处理 array 类型参数时的流程与处理字符串类型参数时十分类似。如图 5-22 所示，名为“arrayToC”的内部函数将被用来处理这些 array 类型的实际参数，该函数在其内部会通过 stackAlloc 函数来分配用于数据交换的共享线性内存空间。该函数仅接收一个参数即待分配内存空间的字节长度，这里直接传入了数组长度。由此可知，Emscripten 现阶段在 ccall 方法中支持传入的数组元素类型只能为 8 位的无符号/有符号整数值类型。


```
// type conversion from js to c
'arrayToC' : function(arr) {
  var ret = stackAlloc(arr.length);
  writeArrayToMemory(arr, ret);
  return ret;
},
```

图5-22 arrayToC函数的代码实现细节

当内存空间分配完毕后，内部函数 `writeArrayToMemory` 将会把数组中的元素值依次写入该内存空间的对应单元中。如图 5-23 所示为该函数的代码实现细节，可以看到 Emscripten 调用了 `HEAP8` 对象内部的 `set` 方法，该方法将会以 8 位有符号整数的形式将数据存储到对应的内存单元中。在这里我们将 `HEAP8` 这类对象称为 Emscripten 运行时环境的内存模型访问符。这些对应于不同数据视图的内存模型访问符，可以按照不同的数据组成形式来访问（读取或写入）共享线性内存中某位置处的二进制数据。

```
function writeArrayToMemory(array, buffer) {
  assert(array.length >= 0, 'writeArrayToMemory array must have a length');
  HEAP8.set(array, buffer);
}
```

图5-23 writeArrayToMemory函数的代码实现细节

如图 5-24 所示为 Emscripten 运行时环境内部支持的常用内存模型访问符类型。由于 MVP 标准下的 `Wasm` 模块最多只能够同时加载和使用一个共享线性内存对象，因此内存模型访问符会直接默认使用当前 `Wasm` 模块初始化时导入的 `WebAssembly.Memory` 内存对象。

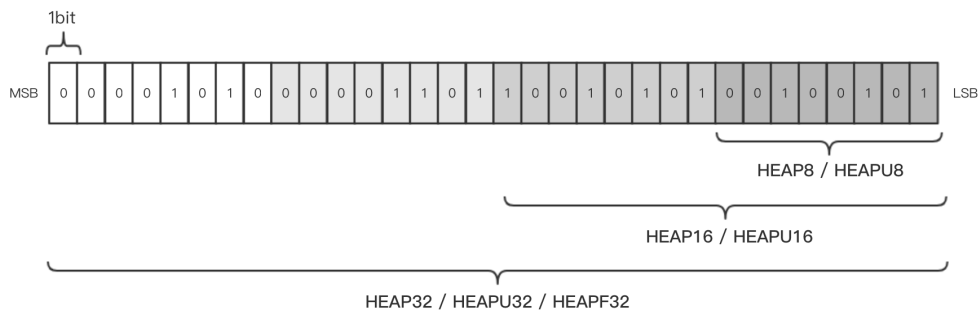


图5-24 Emscripten内存模型访问符类型

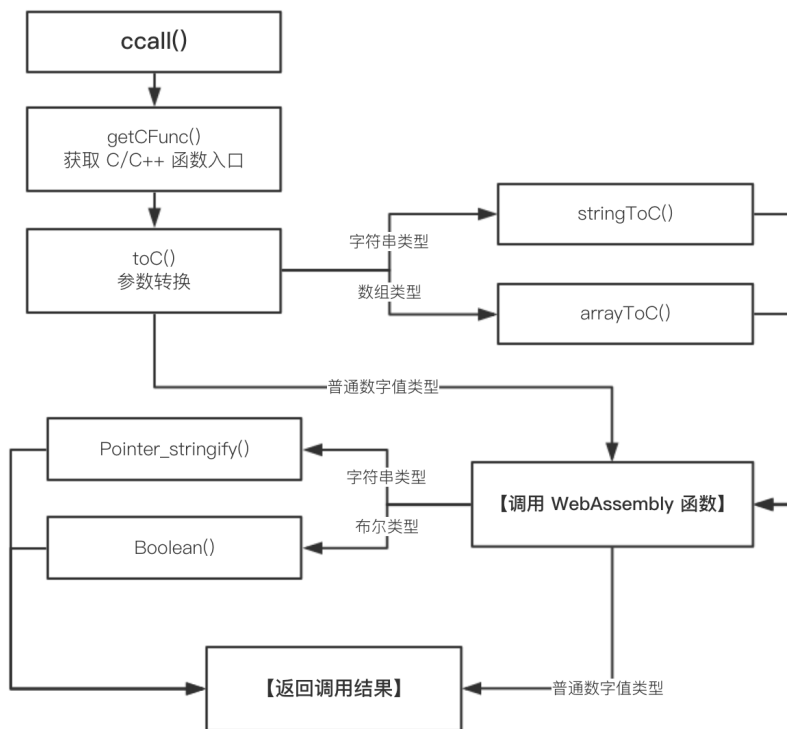
这里一共列出了 7 种类型的内存模型访问符，访问符最后的数字（8/16/32）表示该访问符在处理二进制数据时的数据宽度（数据宽度表示访问符从内存中一次性操作的数据位长度）。比如对于上述 Emscripten 使用的 `HEAP8` 访问符来说，该访问符会以 8 位长数据宽度从内存中某位置处读取数据。以图 5-24 中给出的内存段数据为例，从该内存段最右侧的 `LSB` 位开始，假

设最右侧第一个单元格的内存地址为“0”号索引单元（我们之前介绍过，Wasm 模块的共享内存空间需要通过整数类型的地址来进行索引）。当 HEAP8 访问符作用于该内存位置时，它将会连续读取从“0”号到“7”号共 8 个单元格中的数据，并同时将这些数据按照“8 位有符号整数”的形式进行转换。其对应的二进制数据值 00100101 在经过 HEAP8 访问符的处理后，将会被转换成一个值为“37”的有符号整数。同理，HEAPU 和 HEAPF 访问符则会分别按照无符号整数和浮点数的组成形式，从内存中操作（读取或写入）二进制数据。

至此，我们已经将从上层 JavaScript 环境传递给 Wasm 模块的数组内容写入了共享线性内存段中。现在我们把目光移回到 ccall 方法的后续实现上。由于 arrayToC 函数已经返回了用于存放数组内容的共享线性内存首地址，因此 Emscripten 在调用 C/C++ 代码中定义的 increment 函数时，会将该首地址作为实际参数传递给该函数，这正好与该函数在 C/C++ 代码中定义的形参类型（字符（1byte=8bit）数组的首地址）相符合。当函数处理完数组中的所有元素后，会再次将该数组的对应首地址返回到上层 JavaScript 环境中。这里由于 ccall 方法其本身并不支持将 array 类型作为函数的返回值类型，因此我们需要根据函数返回的数组首地址手动解析出对应共享线性内存空间中的所有数组元素。

这里我们尝试使用 Emscripten 运行时环境提供的另一个常用的内存数据读取函数 Module.getValue，来从共享线性内存中取出数据。该方法一共接收两个参数：第一个参数是待读取数据的起始内存地址；第二个参数是在读取数据时需要使用的数据内存模型（可选值为 i8、i16、i32、i64、float、double 等类型）。所谓的数据内存模型，其实与我们之前介绍的各类内存模型访问符所对应的数据宽度在本质上是一样的。比如我们在该函数中指定的“i8”数据内存模型即对应着 HEAP8 访问符的数据操作方式；“float”数据内存模型则对应着 HEAPF32 访问符的数据操作方式。除此之外，还要注意，当该方法每一次从内存中读取数据后，其第一个参数所代表的起始内存位置都会发生变化，对于连续的内存数据而言，变化的值应该为指定数据内存模型所对应的字节个数（比如“i8”对应 1 个字节，“i64”对应 8 个字节）。从宏观上看，ccall 方法内部的整体代码执行流程如图 5-25 所示。

至此，我们便介绍完了通过 Emscripten “胶水”脚本内置的 ccall 方法来调用 C/C++ 导出函数的基本方式，以及该方法在 Wasm 模块与上层 JavaScript 环境之间传递各种简单及复杂类型参数时的一些细节。实际上，在 emsdk 工具包的 emsdk/emscripten/<version>/src/preamble.js 文件中定义了多种用于进行内存读写和数据编码转换相关操作的内置函数，读者可以在本节内容的基础上自行了解这些方法的具体使用细节。

图5-25 `ccall`方法内部的整体代码执行流程

cwrap

与 `ccall` 方法可以直接调用从 C/C++ 源代码中导出的函数不同, 这里接下来将要介绍的 `cwrap` 方法则会返回 C/C++ 源代码中某个函数对应的 JavaScript 包装函数。通过调用该包装函数, 便可以间接地调用定义在 Wasm 模块中的 C/C++ 函数。如下所示为使用 `cwrap` 方法重写的上一节 Wasm 应用所对应的 JavaScript 主流程代码。

```

post-script.js
__ATPOSTRUN__.push(() => {
  // "doubleSqrt"
  const doubleSqrtResult = Module.cwrap('doubleSqrt', 'number', ['number']);
  // 调用对应的包装函数
  console.log(doubleSqrtResult(400));

  // "capitalize"
  const capitalizeResult = Module.cwrap('capitalize', 'string', ['string']);
  // 调用对应的包装函数

```

```

console.log(capitalizeResult('yhspy'));

// "resetArr2One"
let array = [1, 2, 3, 4];
let increment = Module.cwrap('increment', 'number', ['array', 'number']);
// 调用对应的包装函数
const arrayPointer = increment(array, array.length);
// 定义一个结果集容器
let clearArrResult = [];
for (let i = 0; i < array.length; i++) {
    // 通过 Emscripten 运行时环境的 Module.getValue 函数从共享线性内存中提取数据
    clearArrResult.push(Module.getValue(arrayPointer + i, 'i8'));
}
console.log(clearArrResult);
});

```

如果仔细比较上述代码与之前基于 `ccall` 方法编写的函数调用代码，则会发现这两个方法所接收的参数类型基本相同。唯一不同的是，在 `cwrap` 方法中不需要传递第四个用于表示函数实际调用参数值的“`args`”参数。接下来，我们将从 `cwrap` 方法的代码实现细节入手，来看看它与 `ccall` 方法在使用效果上有哪些不同之处。如图 5-26 所示为 `cwrap` 方法的代码实现细节。

```

function cwrap (ident, returnType, argTypes) {
    argTypes = argTypes || [];
    var cfunc = getCFunc(ident);
    // When the function takes numbers and returns a number, we can just return
    // the original function
    // 判断函数形参类型是否均为数字值；
    var numericArgs = argTypes.every(function(type){ return type === 'number'});
    // 判断返回值是否为数字值（数字+指针）；
    var numericRet = returnType !== 'string';
    // 如果是则直接返回函数实体；
    if (numericRet && numericArgs) {
        return cfunc;
    }
    return function() {
        // 通过柯里化固定一部分参数值；
        return ccall(ident, returnType, argTypes, arguments);
    }
}

```

图5-26 `cwrap`方法的代码实现细节

我们直接观察图 5-26 中该方法定义最后的返回值类型，可以发现实际上 `cwrap` 在其方法内部仍然间接地调用了 `ccall` 方法来执行从 C/C++ 代码中导出的函数。而有意思的地方是，`cwrap` 方法通过函数“柯里化”将用于 `ccall` 方法的部分参数固定住，并将该方法真正的调用过程延迟到由用户主动触发时。从总体上看，这样做的好处就在于，我们可以不用每次都为 `ccall` 方法提

供一长串的类型参数来调用对应的 C/C++ 函数。现在通过 `cwrap` 的函数封装，我们可以提前将导出函数的部分参数固定住，而在真正调用时只需要传递给函数可变的那部分参数即可。

不仅如此，在上面的代码中 `cwrap` 方法有两种返回情况，函数“柯里化”只是其中的一种情况；另一种情况是 `cwrap` 在进行函数“柯里化”之前，会先检测被包装函数的参数与返回值类型，若两者均为数字值类型，则会直接返回对应 C/C++ 函数的导出函数体，即直接从模块“`exports`”对象中导出的 JavaScript 函数。而此时通过 `cwrap` 方法来间接调用 C/C++ 导出函数的效率达到了最高（省去了调用 `ccall` 的步骤）。因此，从总体上看，我们更推荐开发者使用 `cwrap` 方法来间接地调用从 C/C++ 代码中导出的函数。这样做主要有如下两个原因。

- 只需要进行一次函数包装过程，即可在后续的代码中方便地调用。
- 在某些情况下可以提升函数的调用性能。

最后给出 `cwrap` 方法在宏观层面的内部代码执行流程，如图 5-27 所示。

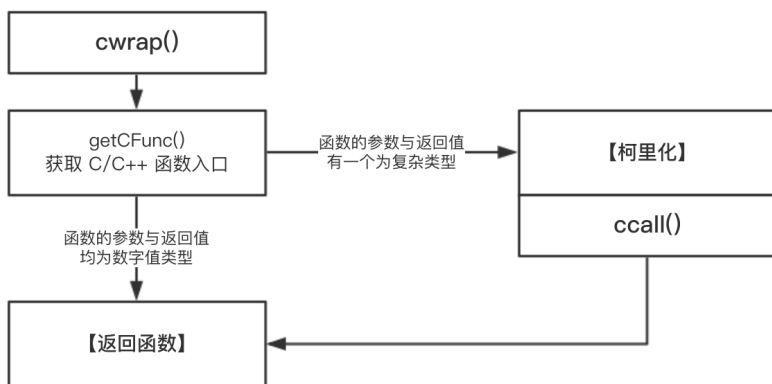


图5-27 `cwrap`方法的内部代码执行流程

至此，我们便介绍完了在 Emscripten “胶水”脚本中通过 `ccall` 和 `cwrap` 方法来调用 C/C++ 导出函数的应用与实现细节。从本质上讲，Emscripten 并没有使用特殊的浏览器接口来访问和调用从 Wasm 模块中导出的 C/C++ 函数，它所做的只是帮助我们将包括参数类型转换、内存访问等在内的 Wasm 模块与上层 JavaScript 环境的交互细节隐藏起来。因此，在构建 Standalone 类型的 WebAssembly 应用时，对于具有复杂参数类型的 C/C++ 函数交互处理方式，可以参考这里的 Emscripten “胶水”脚本的解决思路和实现细节。

而具体是选用 Standalone 类型还是 Dependent 类型的 Wasm 应用，则取决于应用的具体需求和规模。一般来说，对于体量较小并且需要保持高性能的应用，推荐使用 Standalone 的方式

进行构建；而对于规模较大且逻辑较为复杂的应用，则推荐基于 Emscripten 工具链以 Dependent 的方式进行构建。Emscripten 为我们屏蔽了很多需要仔细处理的技术细节，比如自动释放不再使用的共享内存以防止内存泄漏、完善的内存资源访问函数可以有效地防止内存溢出等。而在构建 Standalone 类型的应用时，这部分逻辑就需要我们自己来完成。

5.2.4 在 C/C++ 代码中调用 JavaScript 函数

在上一节内容中，我们主要介绍了如何通过 Emscripten “胶水”脚本在上层 JavaScript 环境中调用从 Wasm 模块中导出的 C/C++ 函数。而在本节的内容中，我们将介绍借助 Emscripten 在 C/C++ 代码中调用上层 JavaScript 函数的几种常用方法。

emscripten_run_script(<code>)

通过使用 `emscripten_run_script(<code>)` 函数，我们可以直接在 C/C++ 代码中引用并执行 JavaScript 代码。在实际调用时，需要将待执行的 JavaScript 代码以字符串指针的形式组合成字符串，并作为参数传递给该函数。示例代码如下：

```
emscripten_run_script.cc
#include <emscripten.h>

int main (int argc, char **argv) {
    // 执行 JavaScript 代码，向浏览器控制台打印内容
    emscripten_run_script("console.log('Hello, WebAssembly!')");
    return 0;
}
```

在上面的代码中，我们通过 `emscripten_run_script` 函数调用了上层 Web 浏览器环境提供的 `console.log` 函数，并向浏览器控制台打印了一段文字内容。本着探索的精神，接下来我们将继续深入到 `emscripten.h` 头文件的内部来查看 `emscripten_run_script` 函数在 `emsdk` 中的代码实现细节。如图 5-28 所示，可以看到，该函数的声明过程被标识为 `extern` 类型，即标志着其函数体被定义在了该文件以外的模块中。

emscripten/system/include/emscripten/emscripten.h	
Line 60 in bf0bdc5	
60	<code>extern void emscripten_run_script(const char *script);</code>

图5-28 `emscripten_run_script`函数在`emsdk`中的代码实现细节

换个思路来想，只要上述 Wasm 应用能够被 `emcc` 正常地编译，那么就说明在对应模块的 WAT（WebAssembly 可读文本代码）文件中一定存在该函数的具体实现代码。接下来，我们先

使用如下命令语句来编译该 Wasm 应用。

```
emcc emscripten_run_script.cc
-s WASM=1
-o emscripten_run_script.html
```

当编译完成后,再通过 Binaryen 工具链提供的 `wasm-dis` 工具来生成该应用内 Wasm 二进制模块所对应的 WAT 可读文本代码文件。接着在该文件中搜索 `emscripten_run_script` 方法的实现细节,这里找到了如图 5-29 所示的一段代码。

```
(import "env" "___unlock" (func $import$23 (param i32)))
(import "env" "_emscripten_memcpy_big" (func $import$24 (param i32 i32
(import "env" "emscripten_run_script" (func $import$25 (param i32)))
(global $global$0 (mut i32) (get_global $import$4))
(global $global$1 (mut i32) (get_global $import$5))
```

图5-29 `emscripten_run_script`函数在模块WAT文件中的引用细节

可以看到, Emscripten 并没有在其内部的 C/C++代码中实现该方法。在实际使用时,我们需要将该方法从 JavaScript 环境导入 (import) Wasm 模块中。下面我们继续在 Emscripten 编译时生成的“胶水”脚本文件中查找关于该函数实现细节的蛛丝马迹。最终我们在“胶水”脚本文件中找到了 `emscripten_run_script` 方法的代码实现细节,如图 5-30 所示。

```
function ___unlock() {}

function _emscripten_run_script(ptr) {
  eval(Pointer_stringify(ptr));
}
```

图5-30 `emscripten_run_script`函数在“胶水”脚本文件中的代码实现细节

该函数首先通过我们之前介绍的 `Pointer_stringify` 方法,根据从 C/C++代码中传入的字符串指针取出了实际待执行的 JavaScript 代码,然后直接通过 JavaScript 语言的标准内置对象 `eval` 执行了这段文本代码。到这里,我们终于弄清了 `emscripten_run_script` 函数在 C/C++代码中执行 JavaScript 代码的整个流程,如图 5-31 所示。

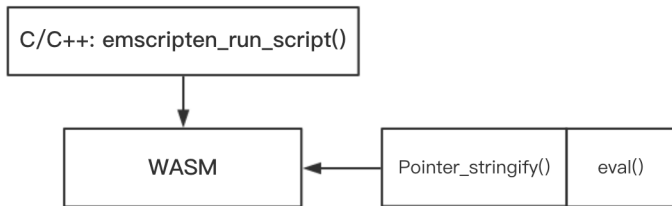


图5-31 `emscripten_run_script`函数在宏观层面的执行流程

总的来看，基于 `eval` 来执行 JavaScript 代码本身并不安全，并且由于浏览器只会通过内部的 JavaScript 解释器来解释执行被写在 `eval` 函数内的代码，因此在没有 JIT 等编译器优化策略的支持下，代码的执行效率并不高。

EM_JS (<ret_type>, <name>, <params>, <body>)

通过 `EM_JS` 这个定义在 `emscripten.h` 头文件中的宏函数，我们可以直接在 C/C++ 代码中定义并直接调用由上层 JavaScript 脚本代码实现的 C/C++ 函数。不仅如此，在实际调用该函数时，还可以从当前的 C/C++ 环境直接向由 JavaScript 实现的函数体内部传递基本类型的数据值。如下为一个简单的应用示例。

```
em_js.cc
#include <emscripten.h>
#include <iostream>

using namespace std;
// 定义一个 add 函数，函数体由 JavaScript 代码实现，声明部分由 C/C++ 实现
EM_JS(int, add, (int x, int y), {
    console.log(x, y);
    return x + y;
});

int main(int argc, char** argv) {
    // 这里直接调用了上面定义好的 add 函数
    cout << add(1, 2) << endl;
    return 0;
}
```

整个 `EM_JS` 宏函数由 4 个参数组成，类似于之前介绍的 `ccall` 和 `cwrap` 方法，这里也需要为将要定义的 C/C++ 函数提供相关的类型信息。其中第一个参数 “<ret_type>” 用于表示该函数的返回值类型，这里可以写入任何 C/C++ 支持的基本数据类型（对于字符串需要使用字符串指针 `char*` 的形式）；第二个参数 “<name>” 为该函数的函数名，由于是作为宏函数的参数，因此这里不需要携带双引号；第三个参数 “<params>” 为以小括号形式表示的函数参数列表，这里可以写入与返回值类似的包含有任何 C/C++ 基本数据类型的形参列表；最后一个参数 “<body>” 为由大括号包裹起来的函数体部分，这里可以直接写入 JavaScript 代码，并在代码中随意使用在参数列表中定义的所有基本类型（数字值、指针）的形参变量。通过 `EM_JS` 宏函数定义的函数，可以直接在随后的 C/C++ 代码中进行使用。

我们通过如下命令来编译该 Wasm 应用。

```
emcc em_js.cc
-s WASM=1
-o em_js.html
```

当应用编译完成后，我们可以在浏览器中运行该应用并看到如图 5-32 所示的结果。其中控制台中的第一行输出内容是在基于 EM_JS 宏函数实现的 add 方法中，通过调用上层 Web 浏览器提供的 console.log 函数打印出来的；第二行输出内容则是在 C/C++ 代码内部直接通过标准库中 cout 对象打印出来的 add 函数调用结果。



图5-32 上述Wasm应用在浏览器中的运行结果

接下来，我们将深入到 emscripten.h 这个头文件中，观察 EM_JS 宏函数在 emsdk 中的代码实现细节。我们可以在 emsdk 工具包所在文件夹下的 emsdk/emscripten/<version>/system/include/emscripten/em_js.h 文件内找到关于该宏函数的完整定义，如图 5-33 所示。

```
emscripten/system/include/emscripten/em_js.h
Lines 1 to 21 in bf0bdc5

1  #ifndef __em_js_h__
2  #define __em_js_h__
3
4  #ifdef __cplusplus
5  #define _EM_JS_CPP_BEGIN extern "C" {
6  #define _EM_JS_CPP_END   }
7  #else // __cplusplus
8  #define _EM_JS_CPP_BEGIN
9  #define _EM_JS_CPP_END
10 #endif // __cplusplus
11
12 #define EM_JS(ret, name, params, ...) \
13   _EM_JS_CPP_BEGIN \
14   ret name params; \
15   __attribute__((used, visibility("default")))\
16   const char* __em_js_##name() { \
17     return #params "<::>" #__VA_ARGS__; \
18   } \
19   _EM_JS_CPP_END
20
21 #endif // __em_js_h__
```

图5-33 EM_JS宏函数在emsdk中的代码实现细节

整个 EM_JS 宏函数的定义按照顺序由如下几个部分组成。

`_EM_JS_CPP_BEGIN / _EM_JS_CPP_END`

`_EM_JS_CPP_BEGIN` 和 `_EM_JS_CPP_END` 这两个宏常量用于组成一个完整的“extern "C">{ ... }”结构。在这两个宏常量之间编写的函数定义将不会受到 C++ 编译器 Name Mangling 过程的影响。

`__attribute__((used, visibility("default")))`

通过使用 `__attribute__((used, visibility("default")))` 属性标签，函数的定义将不会受到编译器 DCE 过程的影响。编译后的函数会被保留在符号表中，并且可以正常地从模块中导出。

`__VA_ARGS__`

`__VA_ARGS__` 预定义宏变量将用于作为“占位符”来替换从 `EM_JS` 宏函数第四个参数（函数体部分）传入函数内部的所有可变参数。

`#` 和 `##`

“`#`”和“`##`”均可以被使用在 C/C++ 代码的预处理过程中，其中“`##`”用于表示直接以连接的方式来拼接两个字面量符号。在上面的代码中，“`__em_js__`”是一个字符串类型，而“`name`”则为宏函数的参数，预处理器在开始执行时会将 `name` 标识直接替换为从宏函数外部传入的函数名实参，“`##`”符号会告知预处理器在拼接这两个字符串时不需要在两者中间插入空格，而是直接以无缝的形式进行连接。

“`#`”符号则主要用于为宏变量的两端添加双引号（`"`）。因此，当调用 `EM_JS` 宏函数时，实际上预处理器会生成与之相对应的 C/C++ 代码段，如下所示。

```
int add (int x, int y);
__attribute__((used, visibility("default")))
const char* __em_js__add() {
    return "(int x, int y)<::>{ console.log(x, y); return x + y; }";
}
```

“`__em_js__`”前缀

`EM_JS` 宏函数在其内部首先声明了函数 `add` 的原型，然后又定义了一个名为“`__em_js__add`”的函数，该函数将会在后续的过程中被 Emscripten 的 JavaScript 编译器后端“Fastcomp”直接使用。如图 5-34 所示，我们可以在 Fastcomp 的 LLVM 后端实现代码中找到与“`__em_js__`”相关的蛛丝马迹。

```
emscripten-fastcomp/lib/Target/JSBackend/JSBackend.cpp
Line 194 in 55fe180

194     const StringRef EM_JS_PREFIX("__em_js_");
```

图5-34 “__em_js_”前缀在具体代码中的封装细节

可以看到，这里“__em_js_”被封装成名为“EM_JS_PREFIX”的字符串引用类型。接下来，我们继续在当前源代码文件中查找与该引用类型有关的代码内容。如图 5-35 所示，在名为“handleEmJsFunctions”的方法中，我们找到了 Fastcomp 处理__em_js__add 函数的主要流程。首先，该方法通过检查函数名是否带有“__em_js_”前缀来确认其是否是是需要处理的目标函数。当检查通过后，该方法会从带有前缀的函数名字符串中截取出我们在 C/C++代码中定义函数时使用的真实函数名（RealName 变量），并连同该函数返回的 JavaScript 函数体定义字符串（其中保存着函数的具体参数列表及函数体实现，两者通过“<::>”符号进行连接）被一起保存在了名为“EmJsFunctions”的字典对象中。

```
emscripten-fastcomp/lib/Target/JSBackend/JSBackend.cpp
Lines 691 to 708 in 55fe180

691     void handleEmJsFunctions() {
692         for (Module::const_iterator II = TheModule->begin(), E = TheModule->end();
693             II != E; ++II) {
694             const Function* F = &*II;
695             StringRef Name(F->getName());
696             if (!Name.startswith(EM_JS_PREFIX)) {
697                 continue;
698             }
699             std::string RealName = "_" + Name.slice(EM_JS_PREFIX.size(), Name.size()).str();
700             const Instruction* I = &*F->begin()->begin();
701             const ReturnInst* Ret = cast<ReturnInst>(I);
702             const ConstantExpr* CE = cast<ConstantExpr>(Ret->getReturnValue());
703             const GlobalVariable* G = cast<GlobalVariable>(CE->getOperand(0));
704             const ConstantDataSequential* CDS = cast<ConstantDataSequential>(G->getInitial
705                 std::string Code = CDS->getAsString();
706             EmJsFunctions[RealName] = escapeCode(Code);
707         }
708     }
```

图5-35 handleEmJsFunctions方法的代码实现细节

如图 5-36 所示，存放在 EmJsFunctions 字典对象中的 EM_JS 元数据，随后会被 Fastcomp 后端以 JSON 的形式组织并输出到对应的中间文本文件中，而该文件中的内容将会在后处理过程中被读取和解析。整个文件将会作为中间载体用于在 LLVM 后端与 Emscripten 的后处理 Python 脚本之间传递信息，而脚本也会根据这些元数据来生成用于连接 Wasm 模块与上层 JavaScript 环境的“胶水”脚本文件。同时，该文件也将用于初始化 Emscripten 运行时环境。

```

emscripten-fastcomp/lib/Target/JSBackend/JSBackend.cpp
Lines 3892 to 3906 in 55fe180

3892     if (EmJsFunctions.size() > 0) {
3893         Out << " , \"emJsFuncs\": {";
3894         first = true;
3895         for (auto Pair : EmJsFunctions) {
3896             auto Name = Pair.first;
3897             auto Code = Pair.second;
3898             if (first) {
3899                 first = false;
3900             } else {
3901                 Out << " , ";
3902             }
3903             Out << "\"" << Name << "\" : \"" << Code.c_str() << "\"";
3904         }
3905         Out << "}";
3906     }

```

图5-36 Fastcomp后端使用EmJsFunctions字典元数据的具体代码

Emscripten 工具链的编译器后处理过程均由 emsdk 内部的众多 Python 脚本完成，如图 5-37 所示为在 emscripten.py 脚本文件中用于处理 EM_JS 元数据的代码细节（函数 create_em_js）。可以看到，在这段代码中，对读入的字符串元数据以“<:;>”符号作为分隔符对其进行了分割（split）操作，分成的左、右两部分分别对应 JavaScript 函数的参数列表和函数体。但由于 JavaScript 是一种弱类型语言，因此这里只获取了参数列表中的参数名部分，相关的类型说明符则被直接丢弃。接下来，根据所获取到的函数名、参数名和函数体，便可以直接通过 function 关键字将其拼接成一个标准的 JavaScript 函数，并将这段字符串代码返回。最后，这段返回的 JavaScript 代码会被直接输出到 Emscripten 的“胶水”脚本文件中。

```

emscripten/emscripten.py
Lines 2016 to 2029 in bf0bdc5

2016     def create_em_js(forwarded_json, metadata):
2017         em_js_funcs = []
2018         separator = '<:;>'
2019         for name, raw in metadata.get('emJsFuncs', {}).items():
2020             parts = raw.split(separator)
2021             assert len(parts) >= 2
2022             args, body = parts[0], separator.join(parts[1:])
2023             args = args[1:-1].split(',')
2024             arg_names = [arg.split()[-1] for arg in args if arg]
2025             func = 'function {}({}){}'.format(name, ','.join(arg_names), body)
2026             em_js_funcs.append(func)
2027             forwarded_json['Functions']['libraryFunctions'][name] = 1
2028
2029         return em_js_funcs

```

图5-37 create_em_js方法的代码实现细节

至此，整个 EM_JS 宏函数的执行流程便十分清晰了，如图 5-38 所示。

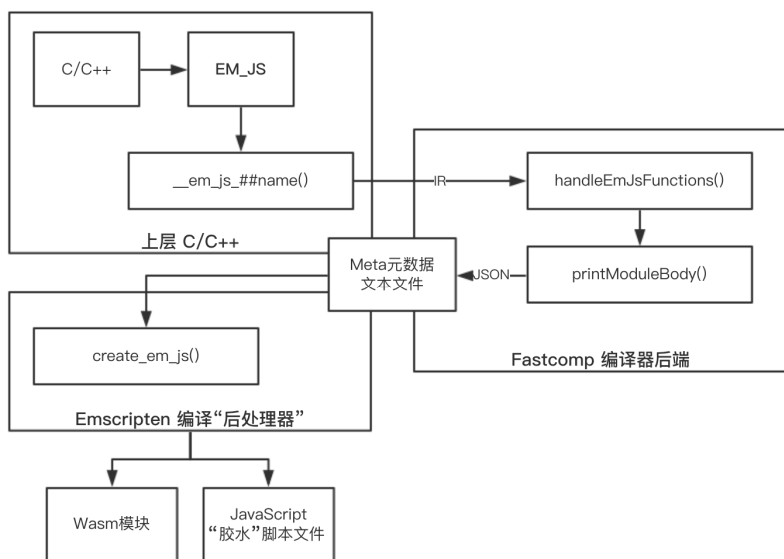


图5-38 EM_JS宏函数在宏观层面的执行流程

总的来说，不论 Emscripten 工具链在其内部怎样组织应用层 C/C++源代码的编写和调用方式，从整体上看，在 C/C++代码中调用 JavaScript 代码这个功能特性都是基于 WebAssembly 的 import 段结构来实现的。通过该段结构，我们可以从上层 JavaScript 环境向模块内部导入仅在模块中声明，并同时调用进行调用的方法，而这些方法的实际代码逻辑（函数体）便可以由上层的 JavaScript 来实现。

EM_ASM(<code>)

EM_ASM(<code>)宏函数与 EM_JS 类似，两者均可用于在 C/C++代码中执行 JavaScript 代码。但不同的是，使用 EM_ASM(<code>)宏函数，我们并不需要通过函数定义来“包装”上层环境中的 JavaScript 代码，而是可以直接编写和执行任意的 JavaScript 代码。如下为一个简单的应用示例。

```

em_asm.cc
#include <emscripten.h>

int main (int argc, char **argv) {
    EM_ASM(
        console.log("Hello WebAssembly! #1");
    );
    EM_ASM(
        console.log("Hello WebAssembly! #2");
    );
}

```

```

);
return 0;
}

```

整个 `EM_ASM(<code>)` 宏函数只接收一个参数，即待执行的 JavaScript 代码片段。我们可以在 `emsdk` 工具包内的 `emsdk/emscripten/<version>/system/include/emscripten/em_asm.h` 头文件中找到关于该宏函数的定义细节，如图 5-39 所示。

```

emscripten/system/include/emscripten/em_asm.h
Lines 152 to 159 in bf0bdc5

152 // Runs the given JavaScript code on the calling thread (synchronously), and return
153 #define EM_ASM(code, ...) ((void)emscripten_asm_const_int(#code _EM_ASM_PREP_ARGS(
154
155 // Runs the given JavaScript code on the calling thread (synchronously), and return
156 #define EM_ASM_INT(code, ...) emscripten_asm_const_int(#code _EM_ASM_PREP_ARGS(_V
157
158 // Runs the given JavaScript code on the calling thread (synchronously), and return
159 #define EM_ASM_DOUBLE(code, ...) emscripten_asm_const_double(#code _EM_ASM_PREP_AR

```

图5-39 EM_ASM宏函数在emsdk中的代码实现细节

这里一共列出了三个与 `EM_ASM` 相关的宏函数定义过程，我们只看其中的第一个宏函数。该宏函数实际上调用了名为“`emscripten_asm_const_int`”的内部方法，并将传递给它的参数传给了该方法，同时通过“(void)”的强制类型转换使得该方法不带有任何返回值。如图 5-40 所示为 `emscripten_asm_const_int` 方法的代码实现细节。这里 `Fastcomp` 在其内部使用了一种特殊的函数注册机制来处理函数的实际调用过程。代码中的 `handleAsmConst` 方法负责处理对应方法调用在 `Wasm` 模块内部产生的调用结果。

```

emscripten-fastcomp/lib/Target/JSBackend/CallHandlers.h
Lines 790 to 797 in 55fe180

790 DEF_CALL_HANDLER(emscripten_asm_const_int, {
791     Declares.insert("emscripten_asm_const_int");
792     return getAssign(CI) + getCast(handleAsmConst(CI, EmAsmOnCallingThread), Type::g
793 })
794 DEF_CALL_HANDLER(emscripten_asm_const_double, {
795     Declares.insert("emscripten_asm_const_double");
796     return getAssign(CI) + getCast(handleAsmConst(CI, EmAsmOnCallingThread), Type::g
797 })

```

图5-40 emscripten_asm_const_int方法的代码实现细节

如图 5-41 所示，`handleAsmConst` 方法将上述 C/C++ 代码对 `emscripten_asm_const_int` 方法的实际调用过程转换为对另外名为“`_emscripten_asm_const_[sig]`”的方法的调用过程。该方法中的“[sig]”占位符表示上层调用函数的返回值签名类型，这里由于为 `int` 类型，因此会使用其缩写字母“`i`”来作为函数后缀名。而函数的调用参数则为通过 `getAsmConstId` 方法获得的函数表索引值。实际上，`Fastcomp` 在其内部维护着一个名为“`AsmConsts`”的字典对象，该对象会以“索引值 - 代码段”的形式来保存所有通过上层 `EM_ASM` 宏函数调用的 JavaScript 代码。

```

emscripten-fastcomp/lib/Target/JSBackend/CallHandlers.h
Lines 763 to 784 in 55fe180

763     std::string handleAsmConst(const Instruction *CI, EmAsmCallType callType) {
764         unsigned Num = getNumArgOperands(CI);
765         std::string Sig;
766         Sig += getFunctionSignatureLetter(CI->getType());
767         for (unsigned i = 1; i < Num; i++) {
768             Sig += getFunctionSignatureLetter(CI->getOperand(i)->getType());
769         }
770         const char *callTypeFunc;
771         switch(callType)
772         {
773             case EmAsmOnCallingThread: callTypeFunc = ""; break;
774             case EmAsmSyncOnMainThread: callTypeFunc = "sync_on_main_thread_"; break;
775             case EmAsmAsyncOnMainThread: callTypeFunc = "async_on_main_thread_"; break;
776             default: llvm_unreachable("Unsupported call type");
777         }
778         std::string func = callTypeFunc + Sig;
779         std::string ret = "_emscripten_asm_const_" + func + '(' + toString(getAsmConstId(CI)) +
780         for (unsigned i = 1; i < Num; i++) {
781             ret += ',' + getValueAsCastParenStr(CI->getOperand(i), ASM_NONSPECIFIC);
782         }
783         return ret + ')';
784     }

```

图5-41 handleAsmConst方法的代码实现细节

如图 5-42 所示为 getAsmConstId 方法的代码实现细节。可以看到,所有通过 handleAsmConst 方法处理的上层 JavaScript 代码都会被存放到 AsmConsts 这个字典对象中。接下来 Fastcomp 对这些代码的进一步处理流程,便与我们之前介绍的 EM_JS 宏函数十分相似。

```

emscripten-fastcomp/lib/Target/JSBackend/JSBackend.cpp
Lines 667 to 689 in 55fe180

667     unsigned getAsmConstId(const Value *V, std::string CallTypeFunc, std::string Sig)
668     {
669         V = resolveFully(V);
670         const Constant *CI = cast<GlobalVariable>(V)->getInitializer();
671         std::string code;
672         if (isa<ConstantAggregateZero>(CI)) {
673             code = " ";
674         } else {
675             const ConstantDataSequential *CDS = cast<ConstantDataSequential>(CI);
676             code = escapeCode(CDS->getAsString());
677         }
678         unsigned Id;
679         if (AsmConsts.count(code) > 0) {
680             auto& Info = AsmConsts[code];
681             Id = Info.Id;
682             Info.Sigs.insert(std::make_pair(CallTypeFunc, Sig));
683         } else {
684             AsmConstInfo Info;
685             Info.Id = Id = AsmConsts.size();
686             Info.Sigs.insert(std::make_pair(CallTypeFunc, Sig));
687             AsmConsts[code] = Info;
688         }
689         return Id;

```

图5-42 getAsmConstId方法的代码实现细节

在 Fastcomp 向外部输出“元数据”文本文件的地方，我们可以看到如图 5-43 所示的这段代码。在这里同样将之前 AsmConsts 字典中存放的 JavaScript 代码，以及与之相对应的索引值，打印到了这个包含用于描述 Wasm 应用特征元数据的文本文件中。因此，后续对这些 JavaScript 代码的处理过程将交由 Fastcomp 的编译器后处理脚本来完成。

```
emscripten-fastcomp/lib/Target/JSBackend/JSBackend.cpp
Lines 3854 to 3862 in 55fe180

3854     Out << "\"asmConsts\": {";
3855     first = true;
3856     for (auto& I : AsmConsts) {
3857         if (first) {
3858             first = false;
3859         } else {
3860             Out << ", ";
3861         }
3862         Out << "\"\" << utostr(I.second.Id) << "\": [\"\" << I.first.c_str() << "\", [";
```

图5-43 Fastcomp向中间元数据文本文件中输出相关内容

如图 5-44 所示，我们在 emscripten.py 这个 Fastcomp 的编译器后处理 Python 脚本中找到了名为“all_asm_consts”的方法，该方法会将元数据文本文件中位于 AsmConsts 字典对象内的 JavaScript 代码通过 function 关键字封装成标准的 JavaScript 匿名函数，这些函数将会在随后的处理流程中被输出到用于连接 Wasm 模块与上层浏览器环境的“胶水”脚本文件中。

```
emscripten/emscripten.py
Lines 731 to 749 in bf0bdc5

731     def all_asm_consts(metadata):
732         asm_consts = [0]*len(metadata['asmConsts'])
733         all_sigs = []
734         all_call_types = []
735         for k, v in metadata['asmConsts'].items():
736             const = asstr(v[0])
737             sigs = v[1]
738             call_types = v[2] if len(v) >= 3 else None
739             const = trim_asm_const_body(const)
740             const = '{ ' + const + ' }'
741             args = []
742             arity = max(map(len, sigs)) - 1
743             for i in range(arity):
744                 args.append('$' + str(i))
745             const = 'function(' + ', '.join(args) + ') ' + const
746             asm_consts[int(k)] = const
747             all_sigs += sigs
748             if call_types: all_call_types += call_types
749             return asm_consts, all_sigs, all_call_types
```

图5-44 all_asm_consts方法的代码实现细节

我们继续探索。如图 5-45 所示，我们发现该 Python 脚本中的 include_asm_consts 方法调用了上述 all_asm_consts 方法。该方法将 all_asm_consts 方法执行后得到的以字符串形式表示的 JavaScript 匿名函数存放到了 asm_consts 变量中。


```

emscripten/emscripten.py
Lines 672 to 678 in b5313f6

672 def include_asm_consts(pre, forwarded_json, metadata, settings):
673     if settings['WASM'] and settings['SIDE_MODULE']:
674         assert len(metadata['asmConsts']) == 0, 'EM_ASM is not yet supported in shared
675
676     asm_consts, all_sigs, call_types = all_asm_consts(metadata)
677     asm_const_funcs = []
678     for s in range(len(all_sigs)):

```

图5-45 include_asm_consts方法的代码实现细节

从图 5-46 中可以看到，该变量随后会被直接拼接到一个名为“ASM_CONSTS”的数组结构中，而被存放在该数组中的匿名 JavaScript 函数也会在 _emscripten_asm_const_[sig] 函数调用时被间接地调用。整个流程正好与我们之前在 Fastcomp 的部分实现代码中看到的 handleAsmConst 方法调用流程相吻合。

```

emscripten/emscripten.py
Lines 698 to 704 in b5313f6

698     asm_const_funcs.append(r'''
699     function _emscripten_asm_const_%s(%s) {
700         %s return ASM_CONSTS[code](%s);
701     }''' % (call_type + asstr(sig), ', '.join(all_args), proxy_to_main_thread, ', '.jo
702
703     asm_consts_text = '\nvar ASM_CONSTS = [' + ',\n'.join(asm_consts) + '];\n'
704     asm_funcs_text = '\n'.join(asm_const_funcs) + '\n'

```

图5-46 EM_ASM宏函数的部分细节调用流程

至此，EM_ASM 宏函数在宏观层面的执行流程如图 5-47 所示。

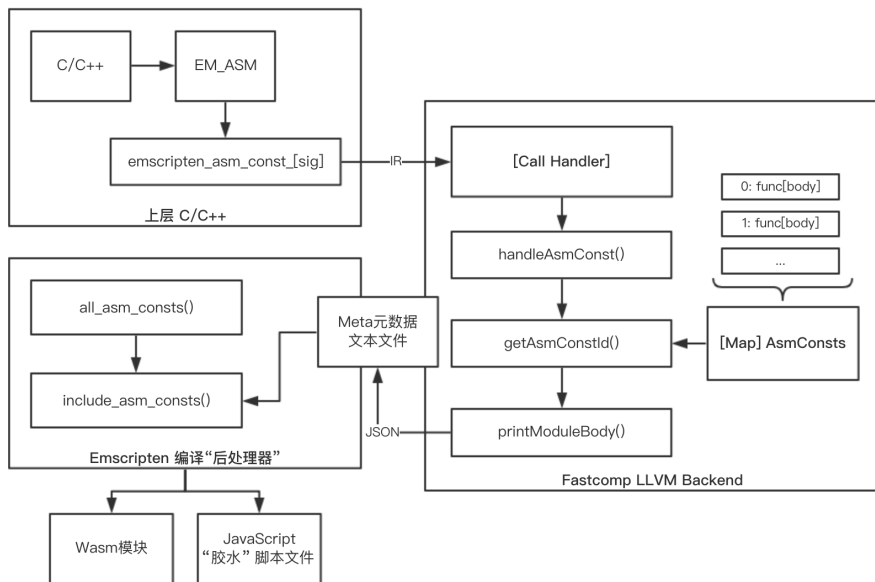


图5-47 EM_ASM宏函数在宏观层面的执行流程

在整个流程中，还需要注意以下两个关键点。

- 实际上，`emscripten_asm_const_int` 方法在调用过程中会向调用者返回一个 `int` 类型的结果值，但该值却由于 `em_asm.h` 文件内该方法被调用时存在强制类型转换符“(void)”而丢失。
- 实际上，在 `handleAsmConst` 方法中，我们可以为 `_emscripten_asm_const_[sig]` 方法传递额外的参数。该方法需要接收的第一个参数是用于在 `ASM_CONSTS` 数组中进行匿名函数查找的索引值，而额外的参数便可以通过上层宏函数的 “`__VA_ARGS__`” 宏常量传入该方法中。这些额外的参数可以帮助我们在上层 JavaScript 匿名函数与 C/C++ 环境之间传递基本类型的数据值。

EM_ASM(<code>, <args>)

`EM_ASM(<code>, <args>)` 宏函数的使用方法与 `EM_ASM` 十分类似，唯一的不同之处在于，通过该宏函数执行的 JavaScript 代码可以直接使用从 C/C++ 环境中传入的基本类型变量值。一个简单的应用示例如下：

```
em_asm.cc
#include <emscripten.h>

int main (int argc, char **argv) {
    // 在 C/C++ 环境中声明一个变量 x
    int x = 100;
    // 可以从该宏函数的第二个参数依次传入定义在 C/C++ 环境中的外部变量
    EM_ASM({
        // 通过变量名 "$0" 来使用传入的 x 变量值
        console.log("This value is from C++: ", $0);
    }, x);

    return 0;
}
```

实际上，该宏函数的实现过程与 `EM_ASM` 宏函数基本一致，两者的调用过程都是基于注册在 Fastcomp 后端的 `emscripten_asm_const_int` 函数进行的。按照上一节中的介绍，传递给该宏函数的除第一个参数以外的其他参数，都会被作为额外的参数值传递给输出到“胶水”脚本文件中的 JavaScript 匿名函数。但需要注意的是，在 JavaScript 脚本中使用从 C/C++ 环境传入的参数时，需要以 “`$[index]`” 的形式进行调用。其中 “[index]” 会从 “0” 开始逐渐递增用于依次表示传入当前函数内的 C/C++ 参数值。

EM_ASM_INT(<code>, <args>)

从 EM_ASM_INT(<code>, <args>)宏函数的名称可以大致猜测到其主要功能。通过该宏函数，我们不仅可以直接从 C/C++环境向 JavaScript 代码中传递变量值，而且在接下来的 C/C++代码中还可以直接使用从该宏函数（JavaScript 环境）返回的值。一个简单的应用示例如下：

```
#include <emscripten.h>
#include <iostream>

using namespace std;

int main (int argc, char **argv) {
    // 整型变量 x 用于接收从 JavaScript 环境返回的值
    int x = EM_ASM_INT({
        console.log('This value is from C++: ' + $0);
        // 将从 C/C++环境传递过来的变量值加 1 后再返回去
        return $0 + 1;
    }, 100);

    cout << x << endl;
    return 0;
}
```

在实现细节上，该宏函数与 EM_ASM 函数基本一致。但需要注意的是，EM_ASM_INT 宏函数只能够从 JavaScript 环境中接收返回来的整型变量值，对于浮点类型则会丢失数据精度。

EM_ASM_DOUBLE(<code>, <args>)

EM_ASM_DOUBLE(<code>, <args>)宏函数的功能与 EM_ASM_INT 的功能类似，区别在于通过该宏函数，我们可以在 C/C++环境中获取从 JavaScript 环境传递过来的浮点类型数据。一个简单的应用示例如下：

```
#include <iostream>
#include <emscripten.h>

using namespace std;

int main (int argc, char **argv) {
    int x = 100;
    // 浮点类型变量 y 用于接收从 JavaScript 环境返回的值
    double y = EM_ASM_DOUBLE({
        // 将从 C/C++环境传递过来的变量值修改后再返回去
    });
}
```

```

    return $0 + 100.1;
}, x);

cout << y << endl;
return 0;
}

```

在实现细节上, 该宏函数与 `EM_ASM_INT` 类似, 区别在于该宏函数使用了名为 `emscripten_asm_const_double` 的内部处理函数, 并且其返回值为 `double` 类型。因此, 对于从 JavaScript 环境传递过来的返回值, 该宏函数将会以双精度浮点数的数据类型来进行解析。

使用 JavaScript 代码定义 C/C++ 函数

除了可以通过 Emscripten 内置在 `emscripten.h` 头文件中的宏函数在 C/C++ 代码中“执行”JavaScript 代码, 我们还可以通过其内部提供的依赖库系统将编写在 C/C++ 代码中的函数声明与对应的 JavaScript 代码实现分离。被分离出来的 JavaScript 代码将会被编写在独立的脚本文件中, 并由 Emscripten 在编译过程中通过依赖分析导入对应的 Wasm 模块。如下为一个简单的应用示例。

```

js_library.cc
#include <iostream>
#include <math.h>

using namespace std;

extern "C" {
    // 声明在外部模块中定义的 custom_add 函数
    extern int custom_add (int x, int y);
}

int main () {
    int x = 10, y = 100;
    // 调用 custom_add 函数
    cout << custom_add(x, y) << endl;
    return 0;
}

```

以上为 C/C++ 部分代码的实现细节, 在这里我们将 `custom_add` 声明为一个外部函数, 并在主函数中进行调用。接下来, 我们将编写被分离出来的 JavaScript 代码, 如下所示。

```

js_library_command.js
mergeInto(LibraryManager.library, {

```

```

custom_add: function(x, y) {
    return x + y;
}
});

```

可以看到，这里我们调用了名为“mergeInto”的函数，该函数的主要功能就是将其第二个参数所对应对象结构内的所有成员函数全部拷贝到第一个参数“LibraryManager.library”所对应的对象结构中。这里的 LibraryManager.library 便是 Emscripten 在其内部维护的一个依赖库对象。在该对象中，存放着 Wasm 应用可以使用到的所有 JavaScript 库函数，这些库函数将会在编译过程中有选择性地被输出到用于连接浏览器与 Wasm 模块的“胶水”脚本文件中。

如图 5-48 所示，我们可以在 JavaScript 脚本文件 emsdk/emscripten/<version>/jsifier.js 中找到 Emscripten 在编译模块时进行的依赖库分析过程。其中变量“ident”为在 Wasm 模块中使用到的外部 JavaScript 库函数名称，每一个库函数所依赖的前置函数都将以函数名的形式被放置在名称为“[ident] + __deps”的数组当中，而 Emscripten 则负责分析相关的依赖并将所需要的依赖函数输出到“胶水”脚本文件中。

```

emscripten/src/jsifier.js
Lines 265 to 270 in bf0bdc5

265     var snippet = LibraryManager.library[ident];
266     var redirectedIdent = null;
267     var deps = LibraryManager.library[ident + '__deps'] || [];
268     deps.forEach(function(dep) {
269         if (typeof snippet === 'string' && !(dep in LibraryManager.library)) warn('missi
270     });

```

图5-48 Emscripten的模块依赖分析过程

现在我们通过如下命令来编译该应用。需要注意的是，这里通过附加一个名为“--js-library”的参数，告知 emcc 另外添加到全局依赖库中的库函数所在的脚本文件。

```

emcc js_library.cc
-s WASM=1
-o js_library.html
--js-library js_library_command.js

```

需要再次明确的是，无论是通过宏函数还是全局依赖库的方式来达到能够在 C/C++代码中直接或间接执行 JavaScript 代码的效果，其基本原理都是基于 WebAssembly 本身的 Import 段结构来实现的。

通过指针在 C/C++代码中调用 JavaScript 函数

接下来我们将尝试通过函数指针的方式，在 C/C++代码中调用编写在独立 JavaScript 脚本文件中的函数。仍然以一个简单的例子作为切入，首先给出的是该应用示例的 C/C++代码。

```
addFunction.cc
```

```
#include <iostream>
#include <emscripten.h>

using namespace std;
// 该函数将接收一个从 JavaScript 环境传递过来的函数指针
extern "C" void EMSCRIPTEN_KEEPALIVE wrapper (int fp) {
    // 定义目标函数类型
    using fpt = void (*)(int);
    cout << "The function pointer is: " << fp << endl;
    // 对传递过来的 int 类型的指针进行类型转换
    fpt f = reinterpret_cast<fpt>(fp);
    // 通过指针调用对应的 JavaScript 函数
    f(7);
    // 清理函数索引表
    EM_ASM({
        Module.removeFunction($0);
    }, f);
}
```

在这段代码中，我们定义了一个名为“wrapper”的函数。该函数的主要功能是负责接收从 JavaScript 环境传递过来的一个整型的函数指针，并根据 JavaScript 代码中原函数的类型将该函数指针转换成特定的指针类型。接下来，我们便可以通过该指针间接地调用定义在 JavaScript 环境中的函数。在代码段的最后，我们通过 EM_ASM_宏函数调用了 Emscripten “胶水”脚本中的 removeFunction 方法，该方法会将函数索引表中对应位置处的函数引用清空。下面我们继续编写 JavaScript 部分的代码。

```
post-script.js
```

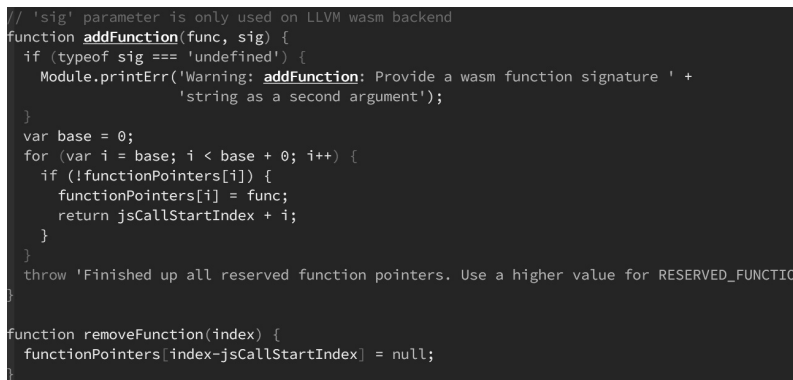
```
__ATPOSTRUN__.push(function () {
    // 通过 Emscripten 内部的 addFunction 向函数索引表中注册一个函数，并返回该函数的函数指针
    var newFuncPtr = Module.addFunction(function(num) {
        console.log(`Hello ${num} from JS!`);
    }, "vi");
    // 直接调用在 C/C++代码中定义的 wrapper 函数
    Module.asm['_wrapper'](newFuncPtr);
});
```

上面代码一共完成两个任务。首先，通过 Module.addFunction 方法将一个 JavaScript 匿名函数注册到全局的函数索引表中，该方法在执行完成后会返回该匿名函数在索引表中的函数指针。然后，调用在 C/C++代码中编写的 _wrapper 函数，并将之前得到的函数指针作为参数传递进去。

代码编写完成后，我们可以通过如下命令来编译该应用。

```
emcc addFunction.cc
--std=c++11
-s WASM=1
-s RESERVED_FUNCTION_POINTERS=20
-s EXTRA_EXPORTED_RUNTIME_METHODS='["addFunction", "removeFunction"]'
-o addFunction.html
--post-js post-script.js
```

现在运行应用，我们可以在浏览器的 Console（控制台）中看到 C/C++ 代码通过函数指针调用对应 JavaScript 函数的结果。下面我们从 Module.addFunction 方法的实现细节入手，来观察整个函数的调用流程。如图 5-49 所示为 Emscripten 为该应用生成的“胶水”脚本文件中 addFunction 和 removeFunction 两个方法的代码实现细节。



```
// 'sig' parameter is only used on LLVM wasm backend
function addFunction(func, sig) {
  if (typeof sig === 'undefined') {
    Module.printErr('Warning: addFunction: Provide a wasm function signature ' +
      'string as a second argument!');
  }
  var base = 0;
  for (var i = base; i < base + 0; i++) {
    if (!functionPointers[i]) {
      functionPointers[i] = func;
      return jsCallStartIndex + i;
    }
  }
  throw 'Finished up all reserved function pointers. Use a higher value for RESERVED_FUNCTION_POINTERS';
}

function removeFunction(index) {
  functionPointers[index-jsCallStartIndex] = null;
}
```

图5-49 addFunction和removeFunction方法代码实现细节

可以看到，用于将函数添加到函数索引表中的 addFunction 方法的实现过程十分简单。在这里名为“functionPointers”的 JavaScript 数组便是函数索引表的主体数据结构，其用于存放将会在 C/C++ 代码中进行函数指针调用的每一个函数的函数体。addFunction 方法在将函数添加到函数索引表后会返回一个整型的数字值，该数字值随后会作为调用指针供 C/C++ 代码使用。

记得我们在之前的编译命令中新增加了一个名为“RESERVED_FUNCTION_POINTERS”的参数，该参数实际上是用来初始化函数索引表的可用大小的。当 Emscripten 在进行模块编译时，会自动根据该参数大小在模块的 Table 段结构中预留出对应个数的用于存放函数引用的单元格。Emscripten 在编译过程中会自动检查在 C/C++ 代码中出现的所有函数指针调用，并根据实际调用的函数指针是否存在对应函数体定义，以及函数指针对应的实际函数签名类型来决定 Wasm 模块内的 call_redirect 指令调用过程。

当模块通过 `call_redirect` 间接地调用存放在 Table 段结构中的函数时，从上层 JavaScript 环境中引入的名为“`jsCall_[sig]`”的函数将会被调用，而实际调用函数的函数名则会根据函数指针类型的不同而带有不同的“`[sig]`”签名后缀（比如 `jsCall_ii` 表示函数形参为两个整型值）。在整个环节的最后，`jsCall_[sig]`函数将会在 JavaScript 环境下通过 Wasm 模块传递过来的函数索引值，从 `functionPointers` 数组中取出对应的目标函数并执行。

基于 Emscripten 工具链，通过函数指针在 C/C++代码中调用 JavaScript 函数的基本流程（宏观层面）如图 5-50 所示。

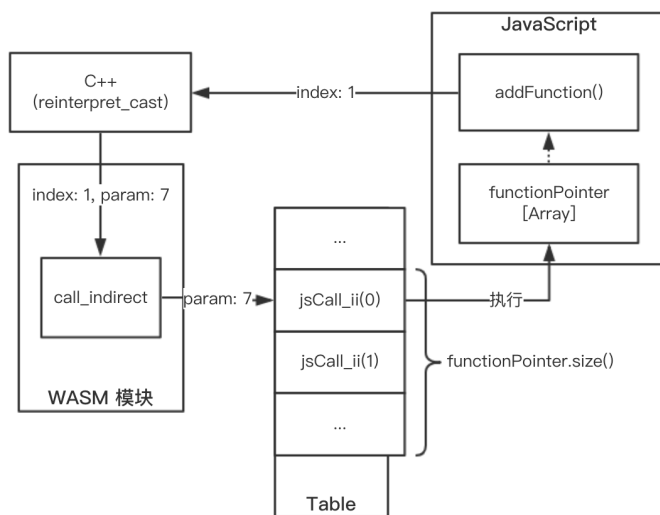


图5-50 通过函数指针在C/C++代码中调用JavaScript函数的基本流程

最后要注意的几点是：相对于 C/C++代码来说，从 JavaScript 环境传递过来的函数指针是不透明的，我们在编写代码时需要通过 `reinterpret_cast` 严格地将这些指针转换为对应 JavaScript 函数的签名类型。因此，专门的 `wrapper` 函数将用于对接专门的某一类 JavaScript 函数指针，而这样可能会损失应用开发的灵活性。并且如果函数指针在转换过程中没有保持严格的函数签名一致性，则可能会引发未定义的错误，让应用调试变得复杂。

第 6 章

基于 Emscripten 的语言关系绑定

提示：本章使用的 emsdk 工具包是 1.38.0 版本。

通过第 5 章的介绍，我们了解到可以通过 Emscripten “胶水”脚本提供的 `ccall` 和 `cwrap` 两个 JavaScript 方法，来直接或间接地调用从 C/C++源代码中导出的函数。这两种方式在具体实现上都是依赖于 WebAssembly 模块的 `export` 段将编译到模块内部的 C/C++函数直接导出到上层 JavaScript 环境中供开发者调用。这样做虽然实现起来十分简单，但也存在如下问题。

- 根据现有的 WebAssembly 标准规定，我们只能通过模块间接地从 C/C++代码向上层 JavaScript 环境中导出函数（`anyfunc`）——这仅有的一种可以直接与 JavaScript 语言本身体系中 `Function` 对象相对应的数据元素。
- WebAssembly 在其标准中仅制定了 4 种基本数据类型（`i32`、`i64`、`f32` 与 `f64`），这些类型可以直接与 C/C++语法标准中的如 `int`、`double` 等基本数据类型进行转换，但对于 `enum` 等复合类型却无能为力。
- 如果需要从 C/C++代码向 JavaScript 环境共享语言层面的数据元素（比如将一个声明在 C/C++代码中的类结构本身共享到 JavaScript 环境中进行使用），在现有的标准下无法直接通过 `Wasm` 模块进行传递，因此也无法通过一些简单的方式来实现。

综上所述，我们能否构建出一种方法可以专门用于在 C/C++和 JavaScript 语言之间进行语法元素上的关系绑定呢？大致的构建思路如图 6-1 所示。首先，我们知道 C/C++是一种语法细节十分丰富的编程语言，比如在声明类的成员方法时，可以为这些方法定义指定 `inline`、`const` 等不同的修饰关键字，这些关键字一方面能够增加代码本身的语义表达能力；另一方面也能够为 C/C++编译器在编译代码时提供更多可用于优化的参考性信息。而 JavaScript 语言则与之正

好相反。由于两者的应用场景不同，JavaScript 并没有在其语言体系中增加过多用于控制代码执行细节的语法结构，而大部分 Web 应用的运行时状态都会由浏览器引擎来自动处理。

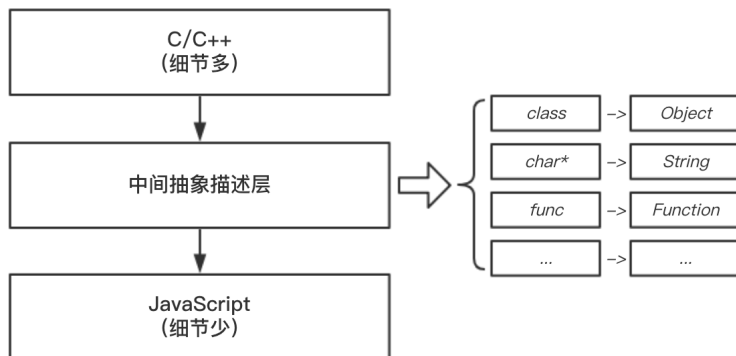


图6-1 在C/C++和JavaScript语言之间进行语法元素关系绑定的基本思路

为了能够从一种语法细节十分丰富的编程语言向另一种语法细节较少的编程语言进行语法元素层面上的数据传递，就需要我们在两种语言之间设置一个专门用来进行语法中转的“中间抽象描述层”结构，以屏蔽两者的语法差异，并绑定可共享的语法结构。在这个结构中，我们需要定义可以在两种语言之间进行共享的所有语法元素类型，并同时提供该元素从源语言到该中间层，以及从该中间层到目标语言的语法转换细节。比如在图 6-1 中，我们在该结构内定义了从 C/C++ 语言的类结构、字符指针等语法元素到 JavaScript 语言体系的具体转换规则。其中，定义在 C/C++ 代码中的类结构将会以 Object 对象的形式被映射到上层 JavaScript 环境中进行使用；而字符指针及函数定义则会分别以 String（字符串）和 Function（函数）对象的形式被映射到上层 JavaScript 环境中。

实际上，根据上面介绍的语言关系绑定流程，Emscripten 已经在其内部为我们整合了两套成熟的解决方案，它们实现语言关系绑定的基本原理以及具体应用方式并不相同。下面我们将对它们两者进行简单介绍。

Embind

Embind 是由 Facebook 工程师 Chad Austin 于 2014 年年底开发的一套专门用于辅助 Emscripten 工具链进行 JavaScript 与 C/C++ 语言之间语法元素关系绑定的中间件子系统。通过 Embind 提供的 EMSCRIPTEN_BINDINGS 宏参数，我们可以直接在 C/C++ 代码中绑定想要“传递”到上层 JavaScript 环境中的语法元素，并且这些元素被映射到 JavaScript 环境中的数据实体还将会被统一注册到“胶水”脚本文件内部的 Module 全局对象中。而 Embind 将源语言与目标语言的语法元素绑定关系，分别实现到了整个子系统对应的部分 C/C++ 文件和 JavaScript 脚本

文件中。

WebIDL Binder

WebIDL 是 W3C 的一套标准规范，该规范在其内部定义了一种独立于具体语言类型的中间“接口定义语言（Interface Definition Language）”，该语言用于描述希望通过第三方语言在 Web 浏览器环境（平台）中实现的接口。Emscripten 在其内部提供了用于将 WebIDL 接口定义代码转换为 JavaScript “胶水”脚本的 WebIDL Binder 工具。但是从 C/C++ 代码到生成对应“.idl”接口定义描述文件的过程，则需要我们根据 WebIDL 的基本语法手动进行“翻译”。另外，相比于 Embind，WebIDL 则是将源语言与目标语言的语法元素绑定关系，以间接的方式全部描述在了以“.idl”为后缀的独立文本文件中。

下面我们将分别介绍如何通过 Embind 中间件和 WebIDL 接口定义语言这两种方式来实现 C/C++ 与 JavaScript 语言的语法元素关系绑定。

6.1 基于 Embind 实现关系绑定

总的来说，基本上在 Emscripten 工具链中通过 Embind 方式实现的所有语言关系绑定都是借助其在头文件中提供的 EMSCRIPTEN_BINDINGS 宏参数来完成的。下面我们通过一个简单的应用示例，从宏观层面完整地了解 Embind 机制的具体实现原理与相关细节。与基于 Emscripten 编写普通的 WebAssembly 应用一样，首先给出的是该应用的 C/C++ 源代码。

```
embind_example.cc
// 引入相关头文件
#include <emscripten/bind.h>
// 与 Embind 相关的宏参数被定义在 emscripten 命名空间中
using namespace emscripten;
// 定义一个函数
int add (int x, int y) {
    return x + y;
}
// 使用 EMSCRIPTEN_BINDINGS 宏参数进行函数绑定
EMSCRIPTEN_BINDINGS(module) {
    function("add", &add);
}
```

在这段代码中，首先定义了一个用于计算两个整数和的 add 函数，然后通过 Embind 中间件提供的 EMSCRIPTEN_BINDINGS 宏参数将该函数绑定到了 JavaScript 环境中。所有与 Embind

机制相关的宏参数都被定义在 `emsdk/emscripten/<version>/system/include/emscripten/bind.h` 头文件中，并通过 `emscripten` 命名空间进行隔离。

使用 `EMSCRIPTEN_BINDINGS` 宏参数进行语言关系绑定一共需要编写两部分代码。第一部分是当前绑定区块的别名，即在上面代码中紧接着宏参数小括号内部的字符串内容。`Embind` 的实现机制规定，我们不能在 C/C++ 代码中定义多个具有相同别名的语言关系绑定区块，因此通过 `EMSCRIPTEN_BINDINGS` 定义的每一个绑定区块都应该具有相对独立的名称和含义。第二部分是位于宏参数后面大括号中的具体绑定内容。在这里将会使用 `Embind` 提供的专门用于绑定 C/C++ 中各类语法元素的子对象和方法，这些对象和方法会按照特定的绑定策略来处理传入其中的待绑定元素。

当 C/C++ 代码编写完成后，我们再来编写上层主业务流程的 JavaScript 代码，并在代码中调用从 C/C++ 绑定到 JavaScript 环境的 `add` 函数。

```
post-script.cc
```

```
__ATPOSTRUN__.push(() => {  
    // 从 C/C++ 环境绑定到 JavaScript 的语法元素将会直接注册到胶水脚本文件的全局 “Module” 对象上；  
    console.log('The result by calling "add": ' + Module.add(1, 2));  
});
```

可以看到，这里并没有通过 `ccall` 或 `cwrap` 方法来调用在 C/C++ 代码中绑定的 `add` 函数。我们直接从 `Module` 全局对象中导出并使用了该函数，而所有需要处理的语法细节绑定信息则全部交由 `Embind` 中间件在 JavaScript 侧的脚本实现代码来完成。

当所有代码编写完成后，我们可以通过如下命令来编译该应用。

```
emcc embind_example.cc  
--bind  
-s WASM=1  
--post-js post-script.js  
-o embind_example.html
```

在上面的命令语句中，我们使用 “`--bind`” 参数让 `emcc` 编译器启用对 `Embind` 中间件的编译时支持。当应用编译完成后，可以在 Web 浏览器中观察其运行结果。

接下来，我们将深入到 `Embind` 机制的内部实现细节中，探索其语言关系绑定过程的具体实现原理。但需要提醒的是，由于 `Embind` 在其代码实现中使用到了 C++ 11 支持的技巧性(tricks)语法，因此这里不会深入到微观的代码实现细节中，而是从宏观层面上来理解 `Embind` 机制的具体实现原理与交互流程。首先，我们将 `Embind` 机制的实现过程分为如下几个主要部分。

Type ID 和 Wire Type

Embind 在其内部为每一种基本的 C/C++ 类型都指定了一个与之对应的唯一数字标识符，即“Type ID”。这些数字标识符将会代替类型本身对应的关键字以方便在后续代码中进行使用。而所有的 C/C++ 数据元素在被映射为 JavaScript 的语法元素前，都会先被转换为其对应的“Wire Type”类型，即 Embind 内部的“中间表示”类型。如表 6-1 所示的是 C/C++ 类型、Wire Type 类型和 JavaScript 类型三者间的对应转换关系（这里只列出部分）。

表 6-1 C/C++ 类型、Wire Type 类型和 JavaScript 类型三者间的对应转换关系

C/C++ 类型	Wire Type	JavaScript 数据类型
int	int	Number
char	char	Number
double	double	Number
std::string	struct { size_t, char[] }*	String
std::wstring	struct { size_t, wchar_t[] }*	String
emscripten::val	_EM_VAL*	任意值类型
class T	T*	Embind Handle

从表 6-1 中可以看到，对于普通的如 int、char 等基本数据类型，其对应的 Embind 内部“中间表示类型”均为其本身；而对于复杂的如 std::string 等字符串类型，其对应的 Wire Type 则是一个复合的结构体类型。在该结构体中，Embind 将一个字符串类型拆分成“长度”和“字符数组”两部分信息。这是由于当我们在 JavaScript 环境中从 Wasm 模块共享线性内存段中提取字符串内容时，需要同时知道该字符串在内存中的偏移位置及对应长度才能够正确地将内容提取出来。表 6-1 中的第 6 种“emscripten::val”类型，可用于直接在 C/C++ 代码中引用上层 JavaScript 环境内的全局对象和方法，这部分内容会放到后面进行介绍。表 6-1 中的最后一种类型为 C++ 中的“类”类型，在这里其对应的 Wire Type 为指向该类的指针类型。该指针会在 JavaScript 环境中被相应的 Embind 处理函数（Handler）处理，进而被转换为某种具体的 JavaScript 数据类型。除表 6-1 中列出的 7 种常见 C/C++ 数据元素类型外，还有如智能指针、重载函数等更为复杂的数据类型，这些类型也都有着其各自对应的 Embind 中间表示类型，即 Wire Type 类型。

语法整合函数

上面介绍的 Type ID 和 Wire Type 主要用来对 C/C++ 代码中的基本数据类型进行包装，而对于诸如函数、结构体等由多种基本数据元素组合而成的复合语法结构，则需要由在 Embind 内部定义的一系列“语法整合函数”来进行处理。比如对于在本节开头给出的示例应用中，Embind 在处理我们定义在 EMSCRIPTEN_BINDINGS 宏参数大括号内的“函数”关系绑定时，会在其

内部调用一个名为“`_embind_register_function`”的语法处理函数来对这些函数绑定关系进行处理。该函数的签名结构如图 6-2 所示。

```
emscripten/system/include/emscripten/bind.h
Lines 77 to 83 in bf0bdc5
77     void _embind_register_function(
78         const char* name,
79         unsigned argCount,
80         const TYPEID argTypes[],
81         const char* signature,
82         GenericFunction invoker,
83         GenericFunction function);
```

图6-2 `_embind_register_function`函数的签名结构

`_embind_register_function` 函数所对应的函数体结构并没有被直接定义在 C/C++代码中，而是间接地通过 JavaScript 代码来实现。因此，当应用运行时，其所依赖的 Wasm 模块需要从上层 JavaScript “胶水”脚本文件中导入该函数对应的函数体部分。从某个角度来看，我们也可以将这些“语法整合”函数视作连接 C/C++和 JavaScript 语言的入口函数。

现在我们把目光移到该函数的参数列表上。该函数一共接收 6 个参数，分别如下。

- **name**: 该参数用于表示需要进行关系绑定的函数名称。
- **argCount**: 该参数用于表示待绑定函数的形参个数。
- **argTypes**: 该参数通过 Type ID 来表示待绑定函数的返回值及参数值类型。
- **signature**: 该参数以字符串的形式表示“invoker 调用器”方法的签名类型，签名顺序为“返回值+参数值”。由于该方法的第二个参数永远为被调用函数的函数指针，因此该位置的签名字符将使用“i”来表示整型。
- **invoker**: 该参数指向一个“调用器”方法。在这里，当我们处于上层 JavaScript 环境中时，需要通过该“调用器”方法来间接地调用绑定的 C/C++函数。
- **function**: 该参数为在 C/C++代码中进行函数关系绑定时传入的原始函数指针。

可以看到，上述第一个参数和最后一个参数是我们在 C/C++代码中通过“function”函数进行函数关系绑定时传入的参数；而其余四个参数则均为 Embind 在其内部根据传入的函数指针自动推导出来的，它们从整体上组成了该次函数关系绑定需要的完整签名信息，这些信息将会被传递到 JavaScript 环境内的 `_embind_register_function` 函数体中进行处理。

在上述示例应用中，通过 Embind 机制进行函数关系绑定的完整内部调用流程如图 6-3 所示。

读者可以在浏览器中通过设置断点的方式来跟踪和理解流程中的每一个步骤。限于篇幅，这里不再展开介绍。

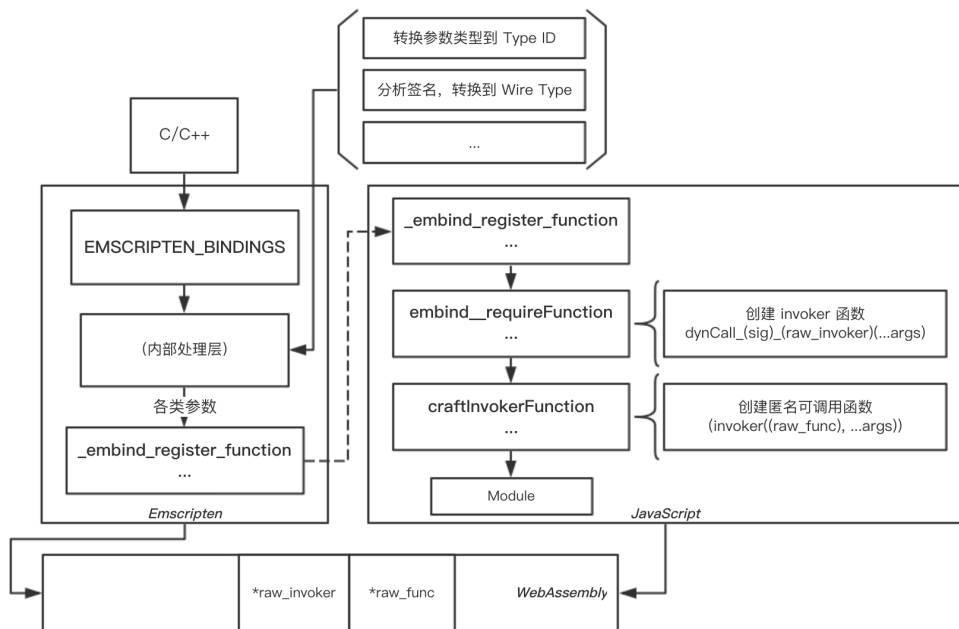


图6-3 基于Embind中间件进行函数关系绑定的完整内部调用流程

这里我们还需要对如下几个地方进行说明，以方便读者的进一步理解。

(1) 由于 Emscripten 并未在 JavaScript 环境中提供类似于“函数表”的对象来直接操作 Wasm 模块中的 Table 段结构，因此，这里以指针形式传入的 invoker 函数便需要通过调用 Emscripten “胶水”脚本中的 dynCall 方法来间接地调用原始函数。而 Embind 在 JavaScript 环境中构建的形式为 “dynCall_(sig)_(raw_invoker)(...args)” 的方法，则通过闭包的语法结构在其内部封装了对 invoker 函数指针的第一层调用过程，如下面的伪代码所示。该方法在实际调用时会返回封装好的 dynCall 方法，该方法会间接地调用存放在模块 Table 段中的 invoker 函数，并将原始函数的指针及参数均传递给该函数，进而间接完成了绑定函数的实际调用过程。

```

(function anonymous(dynCall, raw_invoker) {
  return function dynCall_(sig)_(raw_invoker)(...args) {
    return dynCall(raw_invoker, ...args);
  };
})

```

(2) 关于待绑定函数的实参验证及类型转换过程（Wire Type），这里并没有绘制在上面的

流程图中。事实上，从 C/C++ 代码传入 JavaScript 环境中的形式参数其 Type ID 会被“胶水”脚本文件内名为“whenDependentTypesAreResolved”的方法进行处理。该方法会从已注册的所有基本数据类型列表（registeredTypes）中选择与对应 Type ID 相符合的数据处理函数，来对从该形参位置传入的实参进行诸如类型验证等各种处理过程。

在上面的内容中，我们针对本节开头的示例应用中基于 Embind 进行的函数关系绑定过程，给出了其基本实现原理。实际上，从总体上看，包括函数关系绑定在内的一系列语法元素关系绑定过程都大致相同。接下来，我们将把重点放在基于 Embind 机制对 C/C++ 中各类语法元素进行关系绑定的实际应用方法上，而对代码层面的具体实现细节这里不再深入介绍。

6.1.1 简单类

相比于函数关系绑定，将 C++ 中的“类”类型绑定到 JavaScript 环境中的过程会稍微复杂些。如下为一个简单的应用示例。

```
embind_classes.cc
```

```
#include <emscripten/bind.h>
#include <string>

using namespace std;
using namespace emscripten;

class xClass {
public:
    // 构造函数
    xClass (int x, string y): x(x), y(y) {}

    // 成员函数
    void incrementX () {
        x += 1;
    }

    // 定义成员变量的 [GETTER] 函数
    int getValueX () const {
        return x;
    }

    // 定义成员变量的 [SETTER] 函数
    void setValueX (int val) {
```



```

    x = val;
}

// 静态方法
static string getStringValue (const xClass& instance) {
    return instance.y;
}

private:
    int x;
    string y;
};

EMSCRIPTEN_BINDINGS(module) {
    class_<xClass>("xClass")
        // 绑定构造函数
        .constructor<int, string>()
        // 绑定成员函数
        .function("incrementX", &xClass::incrementX)
        // 绑定私有成员变量的 setter/getter 方法
        .property("x", &xClass::getValueX, &xClass::setValueX)
        // 绑定类的静态方法
        .class_function("getStringValue", &xClass::getStringValue);
}

```

如上面代码所示，我们通过名为“class_”的模板函数以“链式”的方式来实现 C/C++ 中“类”类型的关系绑定过程。这里需要通过该模板函数所返回对象下的各成员函数来分别完成对“类”结构中各种语法元素的绑定过程。比如，`constructor` 模板函数用来绑定“类”的构造函数结构，其模板初始化参数为该构造函数各形式参数的类型标识符；`function` 函数用来绑定“类”结构的多个成员函数；与 JavaScript 中的 `defineProperty` 方法类似，`property` 函数主要用来绑定类中私有成员变量的 `setter` 和 `getter` 访问器方法（由于 JavaScript 并没有私有成员的概念，因此 C/C++ 中定义的私有成员在经过关系绑定后其值将可以被修改）；最后的 `class_function` 函数则专门用于绑定直接定义在“类”结构中的静态函数。

我们可以使用如下 JavaScript 脚本代码，来在上层 JavaScript 环境中使用这个经过关系绑定后的“类”结构。

```

post-script.js
__ATPOSTRUN__.push(() => {

```

```

// 创建一个 xClass 类对象
var xClass = new Module['xClass'](100, "This is a binding class!");
// 打印该对象的私有成员变量 x 的值
console.log(xClass['x']);
// 对该对象的私有成员变量赋值（通过 setter 函数）
xClass['x'] = 98
// 调用该对象的成员函数
xClass.incrementX()
console.log(xClass['x']);
// 调用定义在该对象下的静态方法
console.log(Module['xClass']['getStringValue'](xClass));
// 析构该对象实例，释放 Wasm 共享线性内存空间
xClass.delete();
});

```

对于“类”类型，其内部各语法元素（成员属性/函数、构造函数等）的绑定部分在 JavaScript 环境中的具体使用方法可以参考上面给出的代码，这里不做过多介绍。但需要注意的是，对于所有在 JavaScript 环境中已经使用完毕的“类”实例对象，都需要通过调用其各自的 `delete` 方法来释放它们在 Wasm 模块共享线性内存中占用的空间；否则，随着应用的不断运行，在应用当前的内存空间中创建的类对象实例会越来越多，可能会引发如 OOM 和内存泄漏等问题。

6.1.2 数组与对象类型

Embind 提供了 `value_array` 和 `value_object` 两个模板函数，可以帮助我们将 C/C++ 中的结构体类型与 JavaScript 语言中的数组和对象类型进行绑定，并且这个绑定的作用是双向的。如下为一个简单的应用示例。

```

embind_value_types.cc
#include <emscripten/bind.h>
#include <string>

using namespace emscripten;
// 定义两个结构体
struct Point2f {
    int x;
    int y;
};

struct PersonRecord {

```

```

    std::string name;
    int age;
};

// 绑定到 JavaScript 环境的方法，该方法在内部用到了两种结构体类型
PersonRecord findPersonAtLocation(Point2f &p) {
    PersonRecord pr;
    if (p.x == 0 && p.y == 0) {
        pr = {"YHSPY", 25};
    } else {
        pr = {"anonymous", -1};
    }
    return pr;
}

// 绑定 C/C++ 中的结构体类型
EMSCRIPTEN_BINDINGS(module) {
    // 绑定数组，定义结构体与数组的转换规则
    value_array<Point2f>("Point2f")
        .element(&Point2f::x)
        .element(&Point2f::y);

    // 绑定对象，定义结构体与对象的转换规则
    value_object<PersonRecord>("PersonRecord")
        .field("name", &PersonRecord::name)
        .field("age", &PersonRecord::age);

    // 绑定函数
    function("findPersonAtLocation", &findPersonAtLocation);
}

```

可以看到，在 `findPersonAtLocation` 方法中使用了我们定义在 C/C++ 源代码中的两种结构体类型（`Point2f`/`PersonRecord`）。从整体上看，该方法接收一个结构体引用并同时返回一个新的结构体对象。我们在 `EMSCRIPTEN_BINDINGS` 宏参数内定义了两种结构体与 JavaScript 语法元素的具体绑定规则。通过 `value_array` 方法，我们将 `Point2f` 结构体中名为“x”和“y”的两个元素绑定到了 JavaScript 环境中的一个二元数组上；而通过 `value_object` 方法，将 `PersonRecord` 结构体内名为“name”和“age”的两个属性绑定到 JavaScript 环境中的一个同名对象结构上。

接下来，我们可以通过如下 JavaScript 脚本代码来使用这些绑定好的数据元素。

```
post-script.js
__ATPOSTRUN__.push(() => {
  var person = Module.findPersonAtLocation([0, 0]);
  console.log('Person: ' + person.name + ' - ' + person.age);
});
```

Embind 提供的 `value_array` 和 `value_object` 这两个函数帮助我们定义了从结构体到 JavaScript 语法元素的具体转换规则。可以看到，当从 JavaScript 环境向 `findPersonAtLocation` 函数传递一个二元数组对象（`[object Array]`）时，其在内部会自动按照绑定关系将两个数组值分别对应到 `Point2f` 结构体的两个属性上。同样的，`findPersonAtLocation` 函数返回的结构体对象也会自动根据绑定关系分别对应到一个新的 JavaScript 对象结构（`[object Object]`）的两个同名属性上。不仅如此，Embind 在内部还会自动管理数组与对象结构所占用内存存在整个应用生命周期中的分配和销毁，这让我们在 C/C++ 与 JavaScript 两个环境之间传递和使用复合类型数据变得更加简单。

6.1.3 高级类元素

本节我们将讨论如何基于 Embind 机制来绑定如原始指针、智能指针等与“类”结构相关的复杂语法元素。

原始指针

所谓“原始指针”，其实就是我们在 C 语言中经常使用的普通指针类型。这里之所以将其称为“原始指针”，则是相对于自 C++ 11 标准后提出的“智能指针”而言的。在使用普通的 C 指针时需要我们主动根据应用的执行逻辑来分配和释放内存，因此，在某些情况下，不严谨的代码实现逻辑可能会造成内存泄漏的问题。而由于使用原始指针导致的代码复杂度增加，也是 C/C++ 语言中一个为人诟病已久的问题。

在 Embind 的语言关系绑定过程中，如果需要使用原始指针在 JavaScript 环境中传递对象和值，则必须在绑定对应语法元素时，为绑定对象传递一个名为“`allow_raw_pointers`”的策略属性对象（这里实际上需要传递的是与该策略同名函数的调用返回值），该参数将帮助我们开启 Embind 内部对 C 指针的使用支持。下面给出一个简单的应用示例。

```
embind_advanced_class_raw_pointer.cc
#include <emscripten/bind.h>

using namespace emscripten;

class xClass {
```

```

public:
    xClass(int x) : x(x) {}

    inline int getX () const {
        return x;
    }
    inline void setX (int val) {
        x = val;
    }
private:
    int x;
};
// 该方法在内部使用原始指针来传递对象
xClass* passThrough(xClass* ptr) {
    return ptr;
}

EMSCRIPTEN_BINDINGS(module) {
    class_<xClass>("xClass")
        .constructor<int>()
        .property("x", &xClass::getX, &xClass::setX);
    // 需要添加 allow_raw_pointers() 标志以允许使用原始指针
    function("passThrough", &passThrough, allow_raw_pointers());
}

```

在这段代码中，我们在 `passThrough` 函数内使用了原始指针来进行类对象的传递过程。可以看到，在绑定该函数时，向 `function` 函数传入了 `allow_raw_pointers` 方法的调用返回值来作为其第三个参数。而事实上，在 `Embind` 中间件实现所在的 `bind.h` 头文件中，我们可以清楚地看到该参数直接对应一个名为“Policies”的类型，并且该类型的参数也同样被使用在如类构造函数（`constructor`）绑定、类成员函数（`property`）绑定等其他语法元素的关系绑定过程中。这里以 `function` 函数绑定作为例子，在完成关系绑定后，我们可以使用如下 `JavaScript` 脚本代码来使用该函数。需要注意的是，经由 `passThrough` 函数传入及返回的均是指向同一个类对象的指针。

```

post-script.js
__ATPOSTRUN__.push(() => {
    var xClass = new Module['xClass'](100);
    console.log(xClass['x']); // 100
    // 为对象实体设置一个别名
    var yClass = Module['passThrough'](xClass);

```

```
// 修改指针所指向对象的属性值
yClass['x'] = 0;
console.log(xClass['x']); // 0
console.log(yClass['x']); // 0
xClass.delete();
yClass.delete();
});
```

实际上，由该函数创建生成的 `yClass` 对象与 `xClass` 对象在内存中指向同一个对象实体。因此，当我们改变这两个对象中任意一个对象的属性值时，相应的另一个对象的属性值也会随之发生改变。

外部构造函数

所谓的“外部构造函数”，实际上就是指工厂函数。这是一种十分常见的基于抽象工厂设计模式来创建类对象的方式。抽象工厂设计模式的基本结构如图 6-4 所示。

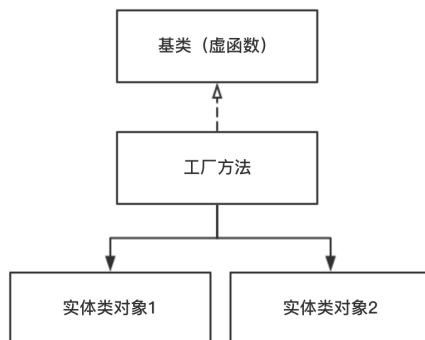


图6-4 抽象工厂设计模式的基本结构

该设计模式的本质是，我们可以在基类中抽象出所有子类都可以公用的方法和属性，而位于中间位置的工厂方法则负责根据传入其内部的实际特征值来创建具有不同特征但属于同一类的类对象实体。举一个简单的例子，比如世界上的所有水果用基类表示，则水果所具有的如颜色、形状等共有属性，以及用于获取这些属性信息的相关方法都可以被放到该基类中，工厂方法会根据传入该方法的具体水果特征（比如名称、类型）来创建出不同的水果对象实体。读者可以上网查阅相关信息来进一步理解该设计模式的整体思路。下面给出一个基于 `Embind` 中间件实现的外部构造函数关系绑定的应用示例。

```
embind_advanced_class_external_constructors.cc
#include <emscripten/bind.h>
#include <string>
```

```

using namespace emscripten;
// 基类
class BaseClass {
    // 子类的一些私有数据
    int age;
    std::string name;
public:
    BaseClass() : age(age), name(name) {};
    // 公有方法
    std::string getName() {
        return name;
    }
};
// 工厂方法
BaseClass* makeClassFactory(int age, std::string name) {
    // 创建子类，并填充父类中的属性
    class xClass : public BaseClass {
    public:
        xClass (int age, std::string name) : BaseClass(age, name) {}
    };
    // 返回一个指向新子类对象的指针
    return dynamic_cast<BaseClass*>(new xClass(age, name));
}
// 绑定外部构造函数
EMSCRIPTEN_BINDINGS(module) {
    class_<BaseClass>("xClass")
        // 这里直接将构造方法绑定到工厂方法，并允许使用原始指针
        .constructor(&makeClassFactory, allow_raw_pointers())
        .function("getName", &BaseClass::getName);
}

```

在这段 C/C++ 代码中，`makeClassFactory` 函数将会作为整个抽象工厂设计模式中的工厂方法，用于向外部输出具有不同特征的实体类对象。这里在通过 `constructor` 方法绑定类的构造函数时，需要将类的实例化过程交由我们自定义的 `makeClassFactory` 函数来进行处理。但无论是通过默认的普通构造函数还是外部构造函数来生成类实例对象，其在 JavaScript 环境中的使用方法均是一致的，代码如下。

```

post-script.js
__ATPOSTRUN__.push(() => {

```

```

var xClass = new Module['xClass'](100, 'YHSPY');
console.log(xClass['getName']());
});

```

智能指针

常用的智能指针一共分为三种类型，这里只介绍 Embind 内部已经支持进行语言关系绑定的 `std::shared_ptr` 和 `std::unique_ptr` 两种。两者的区别是，前者是基于引用计数实现的智能指针，即可以有多个 `share_ptr` 指针同时指向同一块动态分配的内存，并且只有当最后一个指向该内存的 `share_ptr` 指针离开其有效作用域时，这块共享内存才会被完全释放掉；后者则与之相反，当单个 `unique_ptr` 指针离开作用域时，内存便会被释放。

关于智能指针的使用，这里将分为三个部分来介绍，首先介绍如何使用智能指针来维护一个类对象的生命周期。示例代码如下。

```

embind_advanced_class_smart_pointers.cc
#include <emscripten/bind.h>
#include <memory>

using namespace emscripten;

class xClass {
public:
    xClass(int x) : x(x) {}

    int getX() const {
        return x;
    }
    void setX(int val) {
        x = val;
    }
private:
    int x;
};

EMSCRIPTEN_BINDINGS(module) {
    class_<xClass>("xClass")
        // 将智能指针与类对象的创建过程进行绑定
        .smart_ptr_constructor("shared_ptr<xClass>", &std::make_shared<xClass, int>)
        .property("x", &xClass::getX, &xClass::setX);
}

```


在上面的代码中,我们使用 `smart_ptr_constructor` 方法来实现类 `xClass` 的构造函数关系绑定。通过该方法,可以让在 JavaScript 环境中新创建的每一个 `xClass` 类对象都能够以某种特定的共享指针形式来管理其自身的生命周期及相应的资源。该方法一共接收两个参数,其中第一个参数为自定义的共享指针类型名称,我们可以根据所使用的智能指针类型来设置;第二个参数为指向当前“类”类型的某种智能指针的地址,这里直接使用 C++ 11 中的 `make_shared` 模板函数创建了类型为“`shared_ptr`”的共享指针。当然,我们也可以通过编写自定义 `smart_ptr_trait` 模板类的方式来实现自己的智能指针类型。

接下来介绍如何在经过语言关系绑定的 C/C++ 函数中使用指向类对象的智能指针。示例代码如下。

```
embind_advanced_class_smart_pointers.cc
```

```
#include <emscripten/bind.h>
#include <memory>

using namespace emscripten;

class Class {
public:
    Class(int x) : x(x) {}

    int getX () const {
        return x;
    }
    void setX(int val) {
        x = val;
    }
private:
    int x;
};

// 该函数将指向类对象的原始指针转换为智能指针（并非最佳实践，这里仅作为例子供参考）
std::shared_ptr<Class> passThrough(Class *ptr) {
    return std::shared_ptr<Class>(ptr);
}

EMSCRIPTEN_BINDINGS(module) {
    class_<Class>("Class")
        .constructor<int>()
        // 通过该函数，我们可以在经过语言关系绑定的函数中使用指向该类的智能指针
```

```

    .smart_ptr<std::shared_ptr<Class>>("shared_ptr<Class>")
    .property("x", &Class::getX, &Class::setX);

function("passThrough", &passThrough, allow_raw_pointers());
}

```

可以看到，这里在绑定类结构时调用了名为“`smart_ptr`”的模板函数。该函数使得在所有绑定到 JavaScript 环境的函数中，可以直接使用指向该类结构的智能指针来作为函数的参数和返回值。在使用该模板方法时需要为其提供两个参数，其中第一个模板实例化参数为指向该类的智能指针类型（自定义智能指针或标准的 `shared_ptr` 和 `unique_ptr` 类型）；第二个函数调用参数为自定义的共享指针类型名称。

最后介绍 Embind 内部对 `unique_ptr` 智能指针的自动化支持。示例代码如下。

```

embind_advanced_class_smart_pointers.cc

#include <emscripten/bind.h>
#include <memory>

using namespace emscripten;
// 使用 std::unique_ptr 共享指针来传递整数值
std::unique_ptr<int> passToUniquePtr (int v) {
    return std::make_unique<int>(v);
}

EMSCRIPTEN_BINDINGS(module) {
    function("passToUniquePtr", &passToUniquePtr);
}

```

Embind 在内部会自动帮助我们处理 `unique_ptr` 类型的智能指针，因此在代码中无须进行额外的配置即可直接使用。需要注意的是，不同于使用 `shared_ptr` 时 Embind 会自动帮助我们在 JavaScript 环境中返回该指针所指向的对象，这里如果没有通过 `smart_ptr` 或 `smart_ptr_constructor` 方法对指向某“类”类型的 `unique_ptr` 进行注册，那么对于类似上述代码中的 `passToUniquePtr` 方法，其在 JavaScript 环境中被调用时将会直接返回一个指向某数据元素的共享线性内存地址。因此，我们还需要在代码中通过 Emscripten “胶水”脚本提供的 `getValue` 等方法，间接地获取该指针所指向的实际对象值。示例代码如下。

```

post-script.js

__ATPOSTRUN__.push(() => {
    var ptr = Module['passToUniquePtr'](10);
    console.log(Module.getValue(ptr, 'i8'));
});

```

JavaScript 原型链上的非成员函数

借助 Embind，我们可以将一个非成员函数绑定到某个类结构在 JavaScript 环境中所对应对象的原型链（Prototype）上。示例代码如下。

```
embind_advanced_class_non_member_function.cc
#include <emscripten/bind.h>

using namespace emscripten;

class xClass {
public:
    xClass(int x) : x(x) {};

    int getVal (void) const {
        return x;
    }

    void setVal (int val) {
        x = val;
    }

private:
    int x;
};

// 所绑定的非成员函数在调用时，其第一个参数会自动接收一个绑定类的对象引用
void add (xClass &i, int x) {
    i.setVal(i.getVal() + x);
}

EMSCRIPTEN_BINDINGS(module) {
    class_<xClass>("xClass")
        .constructor<int>()
        .function("getVal", &xClass::getVal)
        .function("add", &add);
}
```

被绑定的非成员函数其第一个参数会默认接收当前绑定类的对象引用。接下来，我们可以通过如下脚本代码在 JavaScript 环境中使用上述被绑定的非成员函数。在调用该函数时，其第一个参数会隐式地传递当前调用者的 this 指针，而在实际编写代码时则不需要关心。

```
post-script.js
```

```
__ATPOSTRUN__.push(() => {
  // 创建一个类对象
  var i = new Module['xClass'](100);
  // 该方法的第一个参数会隐式地进行传递
  i['add'](100);
  console.log(i['getVal']());
});
```

在 JavaScript 环境中实现 C++ 派生类

通过 Embind 机制,我们可以在上层 JavaScript 环境中实现在 C/C++ 代码中定义的接口类(纯虚函数),或者覆写非抽象基类中的特定方法。这里将分为两个部分来介绍,首先介绍如何在 JavaScript 环境中实现定义在 C/C++ 代码中的接口类。示例代码如下。

```
embind_advanced_class_deriving_from_js.cc
```

```
#include <emscripten/bind.h>
#include <string>

using namespace emscripten;
// 定义一个接口类,该接口需要由子类来实现
class Interface {
public:
  virtual std::string invoke(const std::string &str) = 0;
};

// 定义一个“胶水”类用来连接 C/C++ 与 JavaScript 代码
class DerivedClass : public wrapper<Interface> {
public:
  EMSCRIPTEN_WRAPPER(DerivedClass);
  std::string invoke(const std::string &str) override {
    // 间接调用在 JavaScript 代码中实现的方法
    return call<std::string>("invoke", str);
  }
};

EMSCRIPTEN_BINDINGS(module) {
  class_<Interface>("Interface")
    // 绑定父类中的抽象接口(纯虚函数)
    .function("invoke", &Interface::invoke, pure_virtual())
    // 通过 allow_subclass 方法向绑定的接口添加两个 JavaScript 方法
```

```
// 即 extend 和 implement，用于实现定义在 C++代码中的接口
.allow_subclass<DerivedClass>("DerivedClass");
}
```

在上面的代码中，我们通过 `wrapper` 模板类构建了一个用于连接 C/C++代码与 JavaScript 环境的“胶水”类。在该类的内部，我们通过调用在 JavaScript 代码中实现的子类接口这种方式来间接地绑定 C++代码中的接口类与 JavaScript 环境中的子类实现过程。而在 `EMSCRIPTEN_BINDINGS` 内部绑定接口类中定义的抽象方法时，我们需要为 `function` 方法提供一个另外的名为“`pure_virtual()`”的策略标志，该标志会标识纯虚函数的绑定过程，并为其提供相应的异常捕获能力。

`Embind` 为我们提供了两个可用于在 JavaScript 代码中实现 C/C++接口的本地函数方法，即 `extend` 和 `implement` 方法。但使用这两个方法的前提是在绑定接口类时，需要通过 `allow_subclass` 方法显式地声明将要在 JavaScript 环境中完成接口类的具体实现过程。接下来，我们便可以借助这两个方法，在 JavaScript 环境中实现 C/C++接口的具体逻辑。

```
post-script.js
```

```
__ATPOSTRUN__.push(() => {
  // 通过 extend 方法来实现子类
  var DerivedClass = Module["Interface"].extend("Interface", {
    // 构造方法（可选）
    __construct: function() {
      // 调用父类的构造函数
      this.__parent.__construct.call(this);
    },
    // 析构方法（可选）
    __destruct: function() {
      // 调用父类的析构函数
      this.__parent.__destruct.call(this);
    },
    // 对接口中纯虚函数的具体实现
    invoke: function(str) {
      return str + " - from 'YHSPY'";
    }
  });

  var instanceByExtend = new DerivedClass();
  console.log(instanceByExtend.invoke("Hello!"));
```

```
// 通过 implement 方法来构造子类
var implementations = {
  invoke: function(str) {
    return str + " - from 'YHSPY'";
  }
};
var instanceByImpelment = Module["Interface"].implement(implementations);
console.log(instanceByImpelment.invoke("Hello!"));
});
```

在这段代码中，首先使用 **extend** 方法完成了 **Interface** 接口类的子类实现过程。与 C/C++ 中继承类的实现过程类似，这里也可以选择性地使用 **__construct** 或 **__destruct** 方法来为该实体类添加相应的构造函数和析构函数。相对于 **extend** 方法而言，**implement** 方法则更适用于不需要构造函数与析构函数的简单接口类。可以看到，这里只需要将与接口类中纯虚函数其签名完全一致的 **JavaScript** 函数以对象结构进行包裹，并传递给从绑定类对象中导出的 **implement** 方法，即可完成对接口类的实现过程。更为方便的是，该方法会直接返回一个已经实例化好的子类对象，这样同时也省去了需要另外再“new”的过程。

接下来介绍如何在 **JavaScript** 代码中覆写定义在 C/C++ 类中的非纯虚函数。示例代码如下。

```
embind_advanced_class_deriving_from_js.cc
#include <emscripten/bind.h>
#include <string>

using namespace emscripten;

class Interface {
public:
  // 定义虚函数（抽象类），已有默认函数实现
  virtual std::string invoke(const std::string &str) {
    return str + " - from 'C++'";
  }
};

class DerivedClass : public wrapper<Interface> {
public:
  EMSCRIPTEN_WRAPPER(DerivedClass);
  std::string invoke(const std::string &str) override {
    return call<std::string>("invoke", str);
  }
}
```

```
};

EMSCRIPTEN_BINDINGS(module) {
    class_<Interface>("Interface")
        // 需要通过 optional_override 方法来创建特殊的 Lambda 函数
        // 以防止 JavaScript 代码与 Wrapper 函数之间产生循环递归调用的问题
        .function("invoke", optional_override([](Interface &self, const std::string &str){
            return self.Interface::invoke(str);
        }))
        .allow_subclass<DerivedClass>("DerivedClass");
}
```

从整体上看，这段代码与前面代码唯一的差别是，当绑定抽象类的非纯虚函数时，我们不能直接向 `function` 方法传递对应函数的指针，而是需要通过 `optional_override` 方法将函数的调用过程封装在一个特殊的匿名函数中并整体传递给 `function` 方法。另外，不同于实现接口类的过程，我们可以在 JavaScript 环境中选择性地覆写或直接使用 `invoke` 函数的默认实现，覆写的具体过程只能以 `extend` 即继承的方式来实现。

绑定 C++ 代码中的派生类

本节将介绍如何将在 C/C++ 代码中实现的子类及其派生父类同时绑定到 JavaScript 环境中。示例代码如下。

```
embind_advanced_class_deriving_from_cpp.cc
#include <emscripten/bind.h>
#include <string>

using namespace emscripten;
// 定义一个基类（父类）
class BaseClass {
public:
    BaseClass() = default;
    // virtual std::string invoke(const std::string &str) = 0;
    virtual std::string invoke(const std::string &str) {
        return str + " - from 'BaseClass'";
    }
};

// 定义继承的子类
class DerivedClass : public BaseClass {
public:
    DerivedClass() = default;
```

```
std::string invoke(const std::string &str) override {
    return str + " - from 'DerivedClass'";
}
};
```

```
EMSCRIPTEN_BINDINGS(module) {
    // 绑定基类
    class_<BaseClass>("BaseClass")
        .constructor<>()
        .function("invoke", &BaseClass::invoke);
    // 绑定子类
    class_<DerivedClass, base<BaseClass>>("DerivedClass")
        .constructor<>()
        .function("invoke", &DerivedClass::invoke);
}
```

在上面的代码中，首先定义了一个名为“BaseClass”的基本类结构，并以该类作为基类（父类）又重新定义了一个名为“DerivedClass”的子类（派生类）结构。实际上，将上述两种类结构绑定到 JavaScript 环境的方法与前面介绍的简单类结构类似，只不过对于子类来说，需要在绑定的过程中通过 `base` 方法来指定该类所对应的父类，而其他元素的绑定过程则没有任何改变。使用这两个类对象的 JavaScript 代码如下。

```
post-script.js
__ATPOSTRUN__.push(() => {
    // 分别创建基类和子类的对象
    var baseClassInstance = new Module["BaseClass"]();
    var derivedClassInstance = new Module["DerivedClass"]();
    // 分别调用两个对象的同一个 invoke 方法
    // 输出: Hello! - from 'BaseClass'
    console.log(baseClassInstance.invoke("Hello!"));
    // 输出: Hello! - from DerivedClass
    console.log(derivedClassInstance.invoke("Hello!"));
});
```

自动向下转型

在某些情况下，Embind 可以帮助我们自动将一个指向基类对象的多态 C++ 指针向下转型到对应的某个子类对象。示例代码如下。

```
embind_advanced_class_auto_downcasting.cc
#include <emscripten/bind.h>
```



```

#include <string>

using namespace emscripten;

class BaseClass {
public:
    BaseClass() = default;
    virtual std::string invoke(const std::string &str) {
        return str + " - from 'BaseClass'";
    }
};

class DerivedClass : public BaseClass {
public:
    DerivedClass() = default;
    std::string invoke(const std::string &str) override {
        return str + " - from 'DerivedClass'";
    }
};

// 该函数的返回值类型将自动向下转型到 DerivedClass 类型的指针
BaseClass* getDerivedInstance() {
    return new DerivedClass();
}

EMSCRIPTEN_BINDINGS(module) {
    class_<BaseClass>("BaseClass")
        .constructor<>()
        .function("invoke", &BaseClass::invoke);
    class_<DerivedClass, base<BaseClass>>("DerivedClass")
        .constructor<>()
        .function("invoke", &DerivedClass::invoke);
    function("getDerivedInstance", &getDerivedInstance, allow_raw_pointers());
}

```

我们可以通过如下 JavaScript 脚本代码来使用上述 C/C++代码中绑定的 `getDerivedInstance` 方法，该方法在实际调用时会返回一个指向子类对象的指针。

```

post-script.js
__ATPOSTRUN__.push(() => {
    var derivedClassInstance = Module['getDerivedInstance']();

```

```
// Output: Hello! - from 'DerivedClass'
console.log(derivedClassInstance.invoke("Hello!"));
});
```

6.1.4 重载函数

Embind 内部并不自动支持基于形参类型的函数重载特性，因此，对于在 C/C++ 代码中定义的重载函数，我们需要通过名为“select_helper”的辅助方法来为每一个重载函数指定其唯一的绑定名称。示例代码如下。

```
embind_overloaded_functions.cc
#include <emscripten/bind.h>
#include <string>

using namespace emscripten;

class OverloadClass {
public:
    OverloadClass() = default;
    // 定义一系列重载函数
    std::string foo () const {
        return "Signature: ()";
    }

    std::string foo (int x) const {
        return "Signature: (int x)";
    }

    std::string foo (int x, int y) const {
        return "Signature: (int x int y)";
    }
};

EMSCRIPTEN_BINDINGS(module) {
    class_<OverloadClass>("OverloadClass")
        .constructor<>()
        // 通过 select_overload 方法为每一个重载函数指定用于函数关系绑定的别名
        .function("foo_v", select_overload<std::string(void) const>(&OverloadClass::foo))
        .function("foo_i", select_overload<std::string(int)>(&OverloadClass::foo))
```

```
.function("foo_ii", select_overload<std::string(int, int)>(&OverloadClass::foo));
}
```

可以看到，我们在名为“OverloadClass”的 C++ 类中定义了多个名称相同但具有不同参数列表的重载函数。这些函数在进行关系绑定时，需要我们依次通过 `select_overload` 模板方法为其指定唯一的可以在 JavaScript 环境中使用的函数名。在使用该方法时，需要为其提供对应的重载函数类型签名及函数指针，Embind 内部会根据函数签名和函数指针所指向的函数位置依次找到每一个重载函数的实体，并为它们分别绑定我们在 `function` 方法中设置的别名。

上述带有别名的重载函数在 JavaScript 环境中的使用方法，与普通类型函数的使用方法没有任何区别。示例代码如下。

```
post-script.js
__ATPOSTRUN__.push(() => {
  var instance = new Module['OverloadClass']();

  console.log(instance['foo_v']()); // Signature: ()
  console.log(instance['foo_i'](10)); // Signature: (int x)
  console.log(instance['foo_ii'](10, 10)); // Signature: (int x, int y)
});
```

6.1.5 枚举类型

Embind 可以同时为 C++ 98 的枚举类型及 C++ 11 中的枚举类提供支持。被绑定的枚举类型或枚举类会以对象（[Object object]）的形式在 JavaScript 环境中使用。示例代码如下。

```
embind_enums.cc
#include <emscripten/bind.h>
using namespace emscripten;
// C++ 98 中的枚举类型
enum OldStyle {
  OLD_STYLE_ONE,
  OLD_STYLE_TWO
};
// C++ 11 中的枚举类
enum class NewStyle {
  ONE,
  TWO
};
```

```
EMSCRIPTEN_BINDINGS(module) {
  // 通过 enum_ 模板方法来绑定枚举类型和枚举类
  enum_<OldStyle>("OldStyle")
    .value("ONE", OLD_STYLE_ONE)
    .value("TWO", OLD_STYLE_TWO);
  enum_<NewStyle>("NewStyle")
    .value("ONE", NewStyle::ONE)
    .value("TWO", NewStyle::TWO);
}
```

绑定枚举类型或枚举类的过程十分简单。通过名为“enum_”的模板方法，我们可以依次为枚举对象中的每一个枚举值都绑定一个与其相对应的“键名”，以方便在 JavaScript 环境中进行引用。

接下来，可以通过如下 JavaScript 代码来使用这些绑定好的枚举值。

```
post-script.js
__ATPOSTRUN__.push(() => {
  console.log(Module['OldStyle']['ONE'].value); // “0”
  console.log(Module['NewStyle']['TWO'].value); // “1”
});
```

6.1.6 基本类型

Embind 可以将 C/C++ 代码中定义的常见基本类型变量值，通过名为“constant”的方法绑定到 JavaScript 环境中进行使用。示例代码如下。

```
embind_constants.cc
#include <emscripten/bind.h>
#include <string>

using namespace emscripten;
// 布尔类型
bool val_bool = true;
// 字符类型
char val_char = 'c';
signed char val_s_char = 'c';
unsigned char val_u_char = 'c';
// 整数值类型
signed char val_s_char = 'c';
unsigned char val_u_char = 'c';
```

```
short val_short = 100;
signed short val_s_short = -100;
unsigned short val_u_short = 100;
int val_int = 100;
signed int val_s_int = -100;
unsigned int val_u_int = 100;
long val_long = 100;
signed long val_s_long = -100;
unsigned long val_u_long = 100;
// 浮点数类型
float val_float = 1.5;
double val_double = 1.5;
// 字符串类型
std::string val_string = "YHSPY";
std::wstring val_wstring = L"于航";

EMSCRIPTEN_BINDINGS(module) {
    // 通过 constant 方法绑定上述基本值变量
    constant("val_bool", val_bool);
    constant("val_char", val_char);
    constant("val_s_char", val_s_char);
    constant("val_u_char", val_u_char);
    constant("val_short", val_short);
    constant("val_s_short", val_s_short);
    constant("val_u_short", val_u_short);
    constant("val_int", val_int);
    constant("val_s_int", val_s_int);
    constant("val_u_int", val_u_int);
    constant("val_long", val_long);
    constant("val_s_long", val_s_long);
    constant("val_u_long", val_u_long);
    constant("val_float", val_float);
    constant("val_double", val_double);
    constant("val_string", val_string);
    constant("val_wstring", val_wstring);
}
```

这里只需要简单地通过 `constant` 方法来为每一个基本变量值绑定一个相对应的 JavaScript 标识符即可。在上层 JavaScript 环境中，我们可以直接通过这些标识符来引用 C/C++ 代码中对应基本变量的实际值。需要注意的是，这里只是值传递。

```
post-script.js
```

```
__ATPOSTRUN__.push(() => {
  console.log(Module['val_bool']); // “true”
  console.log(Module['val_char']); // “99”
  console.log(Module['val_s_char']); // “99”
  console.log(Module['val_u_char']); // “99”
  console.log(Module['val_short']); // “100”
  console.log(Module['val_s_short']); // “-100”
  console.log(Module['val_u_short']); // “100”
  console.log(Module['val_int']); // “100”
  console.log(Module['val_s_int']); // “-100”
  console.log(Module['val_u_int']); // “100”
  console.log(Module['val_long']); // “100”
  console.log(Module['val_s_long']); // “-100”
  console.log(Module['val_u_long']); // “100”
  console.log(Module['val_float']); // “1.5”
  console.log(Module['val_double']); // “1.5”
  console.log(Module['val_string']); // “YHSPY”
  console.log(Module['val_wstring']); // “于航”
});
```

6.1.7 容器类型

如果要在 C/C++ 代码中使用如 `vector`、`map` 等标准模板库（STL）中的容器数据类型，则需要首先通过 `register_vector` 和 `register_map` 这两个 `Embind` 函数来对所使用的具体容器类型进行注册。注册过程会让 `Embind` 在编译代码过程中生成针对特定容器类型数据的 C++ 与 JavaScript “胶水”函数。示例代码如下。

```
embind_container_type.cc
```

```
#include <emscripten/bind.h>
#include <string>
#include <vector>

using namespace std;
using namespace emscripten;

class xClass {
public:
  // 构造方法
```

```

xClass (int x, string y): x(x), y(y) {}

// 使用 vector 向量容器
vector<int> returnVectorData () {
    vector<int> v(10, x);
    return v;
}

// 使用 map 字典容器
map<int, string> returnMapData () {
    map<int, string> m;
    m.insert(pair<int, string>(x, y));
    return m;
}

private:
    int x;
    string y;
};

EMSCRIPTEN_BINDINGS(module) {
    class_<xClass>("xClass")
        .constructor<int, string>()
        .function("returnVectorData", &xClass::returnVectorData)
        .function("returnMapData", &xClass::returnMapData);
    // 注册在代码中使用的容器类型
    register_vector<int>("vector<int>");
    register_map<int, string>("map<int, string>");
}

```

在上面的代码中，我们分别在 `returnVectorData` 和 `returnMapData` 两个函数中使用并返回了相应的 `vector` 向量容器和 `map` 字典容器数据。为了能够让 `Embind` 支持这些容器类型数据的关系绑定过程，我们需要在整段代码结尾处的 `EMSCRIPTEN_BINDINGS` 宏参数内部，通过对应的 `register_vector` 和 `register_map` 模板方法来对它们进行注册。需要注意的是，这里的模板方法在被调用时所使用的初始化参数类型，需要与我们在 C++ 代码中使用的“字典”与“向量”容器元素类型相对应，即有多少种容器类型就需要调用多少次注册过程。

我们可以通过如下 JavaScript 脚本代码来使用上述在 C++ 函数中返回的 `vector` 和 `map` 数据结构。`Embind` 会帮助我们自动地在相应的“胶水”脚本代码中封装好针对这两种容器数据的常

用操作方法，比如通过 `push_back` 方法向一个返回到 JavaScript 环境中的向量容器尾部推入新的元素，或者通过 `set` 方法来设置一个返回到 JavaScript 环境中的字典容器在某个索引位置上的元素值。其他相关方法的使用可以参考如下代码。

```
post-script.js
__ATPOSTRUN__.push(() => {
  var xClass = new Module['xClass'](10, "Key");
  var retVector = xClass.returnVectorData();
  // 获得向量容器的大小
  var vectorSize = retVector.size();
  // 重新设置向量容器中某索引位置上的元素
  retVector.set(vectorSize - 1, 11);
  // 往向量容器尾部推入元素
  retVector.push_back(12);
  // 遍历向量容器
  for (var i = 0; i < retVector.size(); i++) {
    console.log("Vector Value: ", retVector.get(i));
  }
  // 扩增向量容器并设置默认值
  retVector.resize(20, 1);

  var retMap = xClass.returnMapData();
  // 获得字典容器的大小
  var mapSize = retMap.size();
  // 获取字典容器某索引位置上的值
  console.log("Map Value: ", retMap.get(10));
  // 重新设置字典容器某索引位置上的值
  retMap.set(10, "OtherKey");

  xClass.delete()
});
```

6.1.8 转译 JavaScript 代码

Embind 内部提供了一个名为“`emscripten::val`”的类结构，通过该类结构，我们可以在 C/C++ 代码中直接使用暴露在 JavaScript 全局环境内的数据对象和变量。当然，我们也可以直接将上一节介绍的 C/C++ 代码中的基本类型变量，通过该类间接地传递到 JavaScript 环境中进行使用。示例代码如下。


```
embind_val.cc
#include <emscripten/val.h>
#include <emscripten/bind.h>
#include <string>
#include <iostream>

using namespace emscripten;

val getDefaultStrValue (void) {
    // 直接构造的字面量值
    return val("YHSPY");
}

val getDefaultIntValue (void) {
    // 间接通过变量进行构造
    int _t = 10;
    return val(_t);
}

// 操作 DOM 对象
void manipulateDOM (val content) {
    // 获取 JavaScript 环境下的 document 全局对象
    val documentInstance = val::global("document");
    // 判断该对象是否存在
    if (!documentInstance.as<bool>()) {
        std::cout << "No 'window.document' object found!" << std::endl;
        return;
    }
    // 调用该对象并向页面中打印传入的内容
    documentInstance.call<void>("write", content);
    return;
}

void mountTime () {
    val moduleInstance = val::global("Module");
    val DateContext = val::global("Date");
    if (!moduleInstance.as<bool>()) {
        std::cout << "No 'Module' object found!" << std::endl;
        return;
    }
}
```

```
// 创建一个新的 Date 对象
val dateInstance = DateContext.new_();
// 在 Module 全局对象中新建一个 currentTimestamp 属性，并设置该属性值为当前时间戳的数值
moduleInstance.set("currentTimestamp", dateInstance.call<val>("getTime"));
return;
}

EMSCRIPTEN_BINDINGS(module) {
    // 绑定方法
    function("getDefaultStrValue", &getDefaultStrValue);
    function("getDefaultIntValue", &getDefaultIntValue);
    function("manipulateDOM", &manipulateDOM);
    function("mountTime", &mountTime);
}
```

在这段代码中，我们借助 `emscripten::val` 类完成了多种类型的操作。

(1) 我们在两个不同语言环境之间（C/C++与 JavaScript 环境）需要进行数据交换的地方使用 `emscripten::val` 类来包装需要传递的数据值。该类对象同时支持以变量或字面量的形式进行初始化。比如在上面代码的前两个绑定函数中，其返回的数据值通过该类进行了包装，并且函数的返回值也被设置为 `emscripten::val`，而 `Embind` 则会在调用函数时对其参数和返回值自动进行适当的数据类型转换。

(2) 通过 `emscripten::val` 类提供的 `global` 方法，我们可以在 C/C++代码中直接引用暴露在 JavaScript 全局环境中的变量（比如这里引用了上层 Web 浏览器环境中的 `window.document` 全局对象）。所引用的变量将会在 C/C++代码中以同样的 `emscripten::val` 类型进行表示，因此，我们可以进一步通过该类型对象提供的 `call`、`new_` 等方法，在 C/C++代码中间接地对引用的上层 JavaScript 全局变量进行如方法调用、属性绑定等操作。

虽然我们可以通过 C/C++代码间接地调用浏览器中的各类 JavaScript 全局对象，但这并不意味着直接通过 C/C++代码来操作这些对象会有更高的性能。如图 6-5 所示为两种不同的上层 JavaScript 对象调用方式。

可以看到，相对于直接通过上层 JavaScript 代码来操作浏览器的行为，间接地通过 C/C++代码来控制浏览器行为会增加很多无意义的代码处理开销，比如频繁在两种环境（JavaScript 以及 WebAssembly）之间切换产生的上下文开销、数据值类型转换开销，以及“胶水”脚本的执行开销等。因此，是否需要通过 `Embind` 中间件在 C/C++代码中直接进行类似使用 DOM 对象等全局的 JavaScript 对象（变量），则需要我们对项目的整体定位进行评估，并从项目的使用性

能与工程化等角度进行衡量后才能够决定。不过值得庆幸的是，在最新的 Firefox Nightly 浏览器中，从 Wasm 模块内部调用导入的 JavaScript 函数或相反的调用流程其效率已经有了非常高的提升。在某些情况下，甚至比通过 JS 调用 JavaScript 函数的效率还要高。

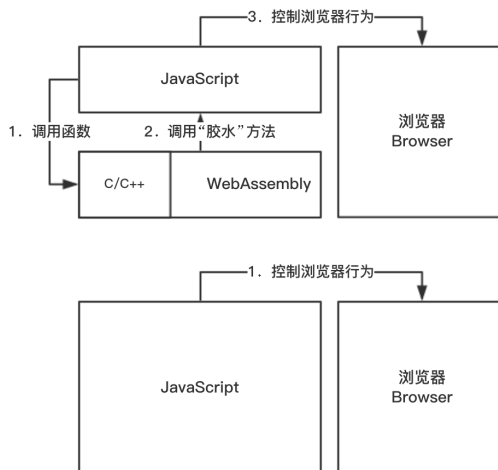


图6-5 两种浏览器全局对象的调用方式

最后，我们通过如下 JavaScript 脚本代码来调用上述绑定好的 C/C++ 函数。

```

post-script.js
__ATPOSTRUN__.push(() => {
  console.log(Module.getDefaultStrValue()); // “YHSPY”
  console.log(Module.getDefaultIntValue()); // “10”
  Module.manipulateDOM("This is from C++!"); // 网页内容变化
  Module.mountTime(); // 调用方法，挂载属性
  console.log(Module['currentTimestamp']); // “1528954718388”
});

```

6.1.9 内存视图

对于某些不透明的数据类型，我们可以将其直接以原始二进制数据的形式在 JavaScript 与 C/C++ 环境之间进行共享，这在某种程度上会省去值复制所带来的开销。这些二进制数据会分别以普通数组和类型数组的形式在 C/C++ 与 JavaScript 环境中存在。示例代码如下。

```

embind_memory_views.cc
#include <emscripten/bind.h>
#include <emscripten/val.h>

```

```
using namespace emscripten;
// 定义一个无符号的字符数组
unsigned char _t [] = {'a', 'b', 'c'};
unsigned char *byteBuffer = _t;
// 获取数组长度
size_t bufferLength = sizeof(_t) / sizeof(unsigned char);

val getBytes() {
    // 通过 typed_memory_view 方法标识类型数组的信息
    return val(typed_memory_view(bufferLength, byteBuffer));
}
EMSCRIPTEN_BINDINGS(module) {
    function("getBytes", &getBytes);
}
```

这里在向 JavaScript 环境返回数组结构时，需要先通过 `typed_memory_view` 方法来标识数组的长度与起始地址，然后将该方法的返回值用 `emscripten::val` 进行包装，最后返回到 JavaScript 环境中。Embind 会将在整个过程中传递的二进制数组内容直接保存到 Wasm 模块的共享线性内存段中。因此，对于多媒体类型的应用，这可以有效地节省其使用的内存空间。

```
post-script.js
__ATPOSTRUN__.push(() => {
    // 这里从 C/C++ 代码中返回的二进制数组将会以 TypedArray 的形式展现
    console.log(Module.getBytes()); // Uint8Array(3) [97, 98, 99]
});
```

6.2 基于 WebIDL 实现关系绑定

相比于 Embind，WebIDL 则使用一种轻量级和标准化的方式来绑定 C/C++ 和 JavaScript 的语法元素。它通过一种特殊的定义语言（WebIDL），将 C/C++ 等第三方语言代码的接口定义关系描述在独立的“.idl”文件中。而 Emscripten 工具链提供的 WebIDL-Binder 脚本，则可以帮助我们对该文件中记录的接口定义关系进行优化，并同时生成与其相对应的 JavaScript “胶水”代码。通过使用这些“胶水”代码，我们便可以在 JavaScript 环境中间接地使用在原始 C/C++ 代码中定义的接口和数据元素。

在 Emscripten 工具链中通过 WebIDL 来实现语言关系绑定分为三个步骤。

- (1) 创建一个用于描述 C/C++ 代码接口组成关系的 WebIDL 文件 (.idl)。
- (2) 使用 WebIDL-Binder 工具生成对应的“胶水”脚本代码。
- (3) 通过 emcc 编译项目，并将在上一步中生成的“胶水”脚本文件作为“--post-js”参数的值以编译到项目中。

我们通过一个简单的应用示例来进行介绍。首先给出的是该应用的 C/C++ 代码。

```
webidl_example.cc
// 定义一个简单的 C/C++ 类
class WebIDL {
public:
    WebIDL (int x, std::string str) : x(x), str(str) {}
    int getValueX (void) const {
        return x;
    }
    const char* getValueStr (void) const {
        return str.c_str();
    }
private:
    int x;
    std::string str;
};
```

接下来，我们使用 WebIDL 语言来描述上述 C/C++ 代码中的类结构以及对外暴露出的接口信息。示例代码如下。

```
webidl_example.idl
interface WebIDL {
    void WebIDL(long x, DOMString str);
    long getValueX();
    [Const] DOMString getValueStr();
};
```

WebIDL 拥有自己的专门的语法结构，这里我们使用 `interface` 关键字来表示在 C/C++ 代码中定义的类结构。该类向外部环境作用域中暴露出三个可以被调用的方法，即类本身对应的构造方法、`getValueX` 和 `getValueStr` 成员方法。接下来，我们需要将这三个方法所对应的签名类型记录在 `interface` 的大括号结构内。WebIDL 在其标准中提供了多种可用的数据类型标识符及扩展属性，但这些数据类型标识符对应的数值取值范围可能与 C/C++ 标准中的同名类型稍有不

同。比如这里在 WebIDL 定义文件中，需要使用“long”标识符来代替在 C/C++代码中用于定义整数的“int”标识符。

当 WebIDL 描述文件编写完成后，我们便可以使用 Emscripten 工具链提供的 WebIDL-Binder 脚本来生成对应的“胶水”脚本文件。命令语句如下：

```
python emsdk/emscripten/<version>/tools/webidl_binder.py webidl_example.idl glue
```

当命令执行完毕后，该脚本会在当前文件夹内生成两个名称分别为“glue.cpp”和“glue.js”的文件。其中 glue.cpp 文件主要用于对 C++源代码文件内的类结构进行处理，在其内部会通过 EMSCRIPTEN_KEEPLIVE 等宏参数将需要关系绑定的方法导出；而 glue.js 文件中的代码则会在 JavaScript 环境中进行相应的初始化工作，并同时定义多种类型的辅助方法，用于维护所导出对象的生命周期。由于 glue.cpp 文件在其内部引用了在 C/C++源代码中声明的类结构，因此我们需要新建一个 C++文件来重新整理其中的构建依赖关系。该文件的内容如下：

```
glue_wrapper.cc
```

```
#include <string>
#include "webidl_example.cc"
#include "glue.cpp"
```

可以看到，这里我们将上述类 WebIDL 声明所在的 C++文件、该类所使用的依赖头文件和 glue.cpp 文件放在了同一个“.cc”文件中。在后续的编译命令中我们将以该文件作为编译的入口文件。接下来，还需要编写上层的 JavaScript 脚本代码来使用已经绑定到 JavaScript 环境中的 WebIDL 类结构。与之前一样，这部分代码仍然被放在单独的 post-script.js 文件中。代码如下：

```
post-script.js
```

```
__ATPOSTRUN__.push(() => {
  // 创建一个新的 WebIDL 对象
  var _t = new WebIDL(10, "YHSPY");
  // 调用该对象上的两个成员方法
  console.log(_t.getValueX()); // "10"
  console.log(_t.getValueStr()); // "YHSPY"
  // 调用对象的析构函数进行垃圾清理
  Module.destroy(_t);
});
```

最后，我们通过 emcc 来编译该 WebAssembly 项目，对应的命令如下：

```
emcc glue_wrapper.cc
-s WASM=1
--post-js glue.js
```

```
--post-js post-script.js
-o webidl_example.html
```

在这里，我们需要将在上一步中通过 WebIDL Binder 脚本生成的 glue.js 文件作为“--post-js”参数的值一起传递给 emcc 编译器。这样编译器便会在编译过程中将该文件内的 JavaScript 脚本代码直接追加到“胶水”脚本文件的尾部，以便在模块初始化时执行。

至此，一个简单的基于 WebIDL 进行语言关系绑定的 WebAssembly 应用便构建完成。在接下来的内容中，我们将针对不同的 C++ 基本语法元素来分别介绍其所对应的 WebIDL 语法，以及绑定元素在 JavaScript 环境中的具体使用方法。

6.2.1 指针、引用和值类型

在 WebIDL 描述文件中，我们可以使用 IDL 语法来分别描述 C/C++ 接口代码中的指针、引用和值三种数据类型。

指针传递

在默认情况下，没有添加任何属性修饰符的标识名称，将用于表示一个以“指针”方式传递的值。示例代码如下：

```
xClass* passThrough(xClass* _t);
```

上述 C/C++ 代码对应的 WebIDL 描述语句为：

```
xClass passThrough(xClass _t);
```

引用传递

对于以 C/C++ 引用形式进行传递的值类型，我们需要在编写对应 WebIDL 代码时通过属性修饰符 “[Ref]” 对其进行修饰。示例代码如下：

```
xClass& passThrough(xClass& _t);
```

上述 C/C++ 代码对应的 WebIDL 描述语句为：

```
[Ref] xClass passThrough([Ref] xClass _t);
```

值传递

对于如 int、double 等基本的 C/C++ 值类型，我们可以直接使用在 WebIDL 语法中定义的如 long、double 等相似的类型修饰符来进行修饰。而对于其他复杂的数据类型来说，如果需要以“传值”的方式来使用它们，则需要通过属性修饰符 “[Value]” 对其进行修饰（在默认情况下为指针传递）。示例代码如下：

```
xClass passThrough(xClass& _t);
```

上述 C/C++ 代码对应的 WebIDL 描述语句为：

```
[Value] xClass passThrough([Ref] xClass _t);
```

综合上述几种数据值传递方式对应的 WebIDL 语法，我们可以通过 Emscripten 将下面这段简单的 C/C++ 代码编译成相应的 Wasm 应用。

```
webidl_value_pass.cc
```

```
class WebIDL {
public:
    // 添加默认的构造函数
    WebIDL () = default;
    WebIDL (int x) : x(x) {}
    int getValueXByValue (void) const {
        return x;
    }
    // 直接返回临时对象
    WebIDL passThroughToValue (WebIDL* i) const {
        return WebIDL(i->getValueXByValue());
    }
    // 返回指向对象的指针
    WebIDL* passThroughByPointer (WebIDL* i) const {
        return i;
    }
    // 返回对象的引用
    WebIDL& passThroughByReference (WebIDL &i) const {
        return i;
    }
private:
    int x;
};
```

上述 C/C++ 代码对应的 WebIDL 描述代码如下：

```
webidl_value_pass.idl
```

```
interface WebIDL {
    void WebIDL();
    void WebIDL(long x);
    long getValueXByValue();
    [Value] WebIDL passThroughToValue(WebIDL i);
```



```
WebIDL passThroughByPointer(WebIDL i);
[Ref] WebIDL passThroughByReference([Ref] WebIDL i);
};
```

接下来，我们可以通过执行 `webidl_binder.py` 脚本，来根据上述 WebIDL 描述文件生成用于绑定 C++语法元素的“胶水”代码（JavaScript/C++）。最后，我们便可以在上层 JavaScript 全局环境中直接使用这些在 C++代码中定义的结构及其成员函数。示例代码如下：

```
post-script.js
__ATPOSTRUN__.push(() => {
  // 生成三个 WebIDL 类对象
  var _t = new WebIDL(10);
  var _k = new WebIDL(11);
  var _u = new WebIDL(12);

  // 调用各种类型的对象“传递”函数
  console.log(_t.passThroughByPointer(_k).getValueXByValue()); // 11
  console.log(_t.passThroughByReference(_u).getValueXByValue()); // 12
  console.log(_t.passThroughToValue(_t).getValueXByValue()); // 10
  Module.destroy(_t);
});
```

6.2.2 类成员变量

对于定义在类结构中的公有（`public`）成员变量，我们需要在对应的 WebIDL 描述文件中通过“`attribute`”关键字来进行标识。示例代码如下：

```
webidl_attribute.cc
class WebIDL {
public:
  WebIDL (int x) : x(x) {}
  // 该类内部一个整型的公有成员变量
  int x;
};
```

上述 C/C++代码对应的 WebIDL 描述代码如下：

```
webidl_attribute.idl
interface WebIDL {
  void WebIDL(long x);
  attribute long x;
};
```

这里直接通过“attribute”关键字来标识该类结构内部的公有成员变量。经过编译后，我们可以通过如下 JavaScript 代码来使用这些公有成员变量。Emscripten 在其内部会为每一个通过“attribute”关键字标识的公有成员变量分配相对应的 Getter/Setter（读取/写入）函数。比如我们可以在上层的 JavaScript 脚本代码中，通过该类结构对象中的 set_x 和 get_x 方法来分别设置和获取成员变量“x”的值。

```
post-script.js
__ATPOSTRUN__.push(() => {
  var _t = new WebIDL(10);
  // 获取公有成员变量的值
  console.log(_t.get_x());
  // 改变公有成员变量的值
  _t.set_x(11);
  Module.destroy(_t);
});
```

6.2.3 常量“const”关键字

对于在 C/C++ 代码中使用“const”关键字标识的各类变量值、参数值和函数返回值，我们都可以在 WebIDL 描述文件中通过“[Const]”属性修饰符或“readonly”关键字来进行相应的标识。其中，属性修饰符“[Const]”主要用于对类成员函数的参数值和返回值进行“const”标识；而“readonly”关键字则主要用于对类结构中的公有成员变量进行标识。示例代码如下：

```
webidl_const.cc
class WebIDL {
public:
  WebIDL (int x) : x(x) {}
  int getValueXByValue (void) const {
    return x;
  }
  // 返回 WebIDL 对象的常量引用
  const WebIDL& passThroughByReference (WebIDL &i) const {
    return i;
  }
  // 常量成员变量
  const int x;
};
```

上述 C/C++ 代码对应的 WebIDL 描述代码如下。可以看到，这里为 passThroughByReference 方法的返回值标识了多个属性修饰符，而对于超过一个的属性修饰符，在编写 IDL 代码时则需

要用方括号 “[]” 将它们包裹起来。

```
webidl_const.idl
interface WebIDL {
    void WebIDL(long x);
    long getValueXByValue();
    [Ref, Const] WebIDL passThroughByReference([Ref] WebIDL i);
    readonly attribute long x;
};
```

经过编译后，我们可以通过如下 JavaScript 脚本代码来使用该类结构。这里将 WebIDL 对象 “_t” 的原型链展开后，可以看到，Emscripten 并没有为标识了 “readonly” 的成员变量 “x” 设置其对应的 “set_x” 函数，因此我们只能够读取该变量的值，而不能对其进行修改。

```
post-script.js
__ATPOSTRUN__.push(() => {
    var _t = new WebIDL(10);
    console.log(_t.getValueXByValue()); // 10
    console.log(_t.passThroughByReference(_t).getValueXByValue()); // 10
    Module.destroy(_t);
});
```

6.2.4 命名空间

对于被定义在 C/C++ 命名空间中的类结构，在其对应的 WebIDL 描述文件内，我们需要使用 WebIDL 语法中的 “Prefix” 关键字来明确指出这些类结构的具体声明位置，即其所在命名空间的名称。示例代码如下：

```
webidl_namespace.cc
// 创建一个名为 “WebAssembly” 的命名空间
namespace WebAssembly {
    // 定义类结构
    class WebIDL {
    public:
        WebIDL (int x) : x(x) {}
        int getValueXByValue (void) const {
            return x;
        }
        int x;
    };
}
```

如下所示，实际上“Prefix”关键字指出了在调用相应接口实体时需要添加的前缀信息。这里的“WebAssembly::”，与在 C++代码中调用标准库方法时所使用的“std::”命名空间前缀十分相似，因此我们也可以按照同样的方式来进行理解。

```
webidl_namespace.idl
```

```
[Prefix="WebAssembly::"]  
interface WebIDL {  
    void WebIDL(long x);  
    long getValueXByValue();  
    attribute long x;  
};
```

在前面的 C/C++代码中，虽然我们将 WebIDL 类结构定义在了名为“WebAssembly”的命名空间中，但是经过 WebIDL 中间层的转译与 Emscripten “胶水”代码的处理，最终可以直接在上层 JavaScript 全局环境中使用这些定义在命名空间中的类结构。从下面的代码可以看出，WebIDL 类的实际使用方式与前面我们给出的示例没有任何差别。

```
post-script.js
```

```
__ATPOSTRUN__.push(() => {  
    var _t = new WebIDL(10);  
    console.log(_t.getValueXByValue()); // 10;  
    Module.destroy(_t);  
});
```

6.2.5 运算符重载

对于在 C++代码中定义的重载运算符来说，我们需要在其对应的 WebIDL 描述文件中，通过“Operator”关键字来指定该运算符所对应的上层 JavaScript 环境中的自定义函数名。Emscripten 会使用该函数来模拟运算符重载函数的具体功能。示例代码如下：

```
webidl_overload.cc
```

```
class WebIDL {  
public:  
    WebIDL (int x) : x(x) {}  
    int getValueXByValue (void) const {  
        return x;  
    }  
}  
// 对运算符“[]”的重载过程
```

```
int operator[](int x) {
    return x * 2;
}
int x;
};
```

在 JavaScript 语言规范中并没有建立对函数或运算符的重载机制，因此，这里 Emscripten 需要为在 C++代码中定义的每一个重载运算符都指定一个唯一的函数名，而该函数名将用于在上层 JavaScript 环境中间接地调用对应运算符的重载函数。比如在下面的 WebIDL 描述代码中，我们为上述 C++代码中的“[]”运算符重载函数指定了一个 `doubleNum` 别名。在这里我们可以把该别名看作一个普通的函数名，在 WebIDL 中也同样需要为该“函数”指定相应的返回值及参数值类型。而这部分内容需要与在 C++代码中对应的运算符重载函数签名保持一致。

```
webidl_overload.idl
interface WebIDL {
    void WebIDL(long x);
    long getValueXByValue();
    [Operator="[]"] long doubleNum(long x);
    attribute long x;
};
```

与普通的类成员函数一样，在上层 JavaScript 环境中，我们可以直接通过在 WebIDL 代码中为相应重载运算符绑定的自定义函数名来间接地使用该重载运算符的功能。

```
post-script.idl
__ATPOSTRUN__.push(() => {
    var _t = new WebIDL(10);
    console.log(_t.doubleNum(_t.getValueXByValue())); // 20
    Module.destroy(_t);
});
```

6.2.6 枚举类型

对于在 C/C++代码中使用的“enum”（枚举）类型，我们需要分成三种使用场景来分别介绍其对应的 WebIDL 代码表达方式。

定义在全局环境中的“enum”类型

我们所说的“裸露”枚举类型，就是指那些直接定义在全局环境中的“enum”类型，该类型在 C/C++代码及其所对应的 WebIDL 描述文件中的写法基本相同。示例代码如下：

```
webidl_enum.cc
// 定义一个名为“xEnum”的枚举类型
enum xEnum {
    enum_val1,
    enum_val2
};
```

上述 C/C++ 代码对应的 WebIDL 描述代码如下：

```
webidl_enum.idl
enum xEnum {
    "enum_val1",
    "enum_val2"
};
```

可以看到，在 WebIDL 语法中，我们需要将该枚举类型中包含的枚举值通过“双引号”包裹起来。

我们可以通过如下的 JavaScript 脚本代码来使用这些被定义在 C/C++ 代码中的枚举值。而 Emscripten 则会自动将这些枚举结构中的枚举值逐个“挂载”到运行时环境中的 Module 全局对象上。

```
post-script.js
__ATPOSTRUN__.push(() => {
    console.log(Module['enum_val1']); // 0
    console.log(Module['enum_val2']); // 1
});
```

定义在命名空间中的“enum”类型

与定义在全局环境中的“enum”类型有所不同，在为定义于命名空间中的“enum”类型编写其对应的 WebIDL 代码时，需要以固定的格式将命名空间的信息整合到独立的“enum”定义结构中。示例代码如下：

```
webidl_enum.cc
// 定义一个名为“yEnum”的枚举类型
namespace WebAssembly {
    enum yEnum {
        ns_enum_val1 = 10,
        ns_enum_val2
    };
};
```

```
// 绑定类型（必须存在）
typedef WebAssembly::yEnum WebAssembly_yEnum;
```

上面的 C/C++ 代码对应的 WebIDL 描述代码如下。可以看到，这里需要分别在“enum”类型的名称及其内部各枚举值的名称中，以特定的格式“混入”与命名空间名称（WebAssembly）相关的信息。需要注意的是，当在 WebIDL 中以下画线（WebAssembly_yEnum）的形式来连接命名空间和枚举类型的名称时，需要在对应的 C++ 代码中进行相应的类型定义（typedef）。通过类型定义，我们会将该名称直接指向对应命名空间中的枚举类型实体。

```
webidl_enum.idl
enum WebAssembly_yEnum {
    "WebAssembly::ns_enum_val1",
    "WebAssembly::ns_enum_val2"
};
```

这样，通过 JavaScript 全局环境中的 Module 对象，我们便可直接使用定义在枚举类型中的枚举值。

```
post-script.js
__ATPOSTRUN__.push(() => {
    console.log(Module['ns_enum_val1']); // 10;
    console.log(Module['ns_enum_val2']); // 11;
});
```

定义在类结构中的“enum”类型

类似的，对于定义在类结构中的枚举类型，也需要按照特殊的形式来编写其对应的 WebIDL 描述代码。示例代码如下：

```
webidl_enum.cc
// 定义一个名为“xClass”的类结构
class xClass {
public:
    xClass(int x) : x(x) {}
    // 在类结构内部定义了一个名为“zEnum”的枚举类型
    enum zEnum {
        c_enum_val1 = 20,
        c_enum_val2
    };
private:
    int x;
};
```

```
// 绑定类型（必须）
typedef xClass::zEnum xClass_zEnum;
```

上面的 C/C++ 代码对应的 WebIDL 描述代码如下。这里需要将 xClass 类结构的定义过程和 zEnum 枚举类型的定义过程分成两个独立的部分来进行描述。其中枚举类型的描述方式与上一节中定义在命名空间中的枚举类型其描述方式完全相同，只不过这里需要将命名空间的名称替换为对应的类名称。

```
webidl_enum.idl
// 枚举类型的描述代码
enum xClass_zEnum {
    "xClass::c_enum_val1",
    "xClass::c_enum_val2"
};
// 类结构的描述代码
interface xClass {
    void xClass(long x);
};
```

枚举值在上层 JavaScript 环境中的使用方式与之前的示例没有太大的区别。唯一需要注意的是，这里可以直接通过类名（xClass）来调用定义在类内部的枚举值，而不再需要另外创建新的类对象。这与 C/C++ 中枚举类型的特性保持着高度一致。

```
post-script.js
__ATPOSTRUN__.push(() => {
    console.log(Module['xClass']['c_enum_val1']); // 20
    console.log(Module['xClass']['c_enum_val2']); // 21
});
```

6.2.7 接口类

与 Embind 中间件一样，基于 WebIDL，我们同样也可以使用 JavaScript 代码来实现定义在 C++ 代码中的抽象类（虚函数）接口。示例代码如下：

```
webidl_subclass.cc
class xClass {
public:
    // 声明一个纯虚函数，该接口将由 JavaScript 代码来实现
    virtual int virtualFunc (int x) = 0;
    // 该函数负责间接“调用”纯虚函数
    static int runVirtualFunc(xClass *self, int x) {
```



```

    return self->virtualFunc(x);
}
};

```

可以看到，在上面的 C++ 代码中，我们在 `xClass` 类内部声明了一个名为“`virtualFunc`”的纯虚函数，该函数的函数体将会放在后续的上层 JavaScript 环境中来进行实现。另外一个名为“`runVirtualFunc`”的函数则在其内部间接地调用了 `virtualFunc` 接口，该函数将会被作为测试函数放到 JavaScript 环境中使用。接下来，为了能够在 JavaScript 环境中实现定义在抽象类中的接口，我们需要使用特殊的 WebIDL 语法结构来绑定该抽象类与其对应的实现类。

```

webidl_enum.idl
// 对接口类对应实现类的描述
[JSImplementation="xClass"]
interface ImplXClass {
    void ImplXClass();
    long virtualFunc (long x);
};
// 对接口类的描述
interface xClass {
    long virtualFunc (long x);
    static long runVirtualFunc(xClass self, long x);
};

```

如上所示，这里主要通过“`JSImplementation`”关键字来为 `xClass` 接口类指定其所对应的 JavaScript 实现类 `ImplXClass`。在该类的 WebIDL 描述代码中，我们需要将所有待实现的抽象方法（纯/虚函数）全部标记在这里。除此之外，对应于接口类本身的 WebIDL 描述代码也需要被写入相应的“`.idl`”描述文件中。接下来，我们便可以在上层 JavaScript 代码中来实现定义在 C++ 代码中的抽象类接口。示例代码如下：

```

post-script.js
__ATPOSTRUN__.push(() => {
    var _t = new ImplXClass();
    // 实现接口
    _t.virtualFunc = function(x) {
        return 2 * x;
    };
    // 间接调用实现类的 virtualFunc 方法
    console.log(_t.runVirtualFunc(_t, 10)); // 20;
    console.log(Module['xClass'].prototype.runVirtualFunc(_t, 10)); // 20;
});

```

在这段代码中，我们需要为 `ImplXClass` 实现类添加对应的 `virtualFunc` 方法。从某种程度上讲，我们可以将该 `JavaScript` 类结构视为 `C++` 代码中 `xClass` 抽象类的实现子类。因此，通过该类对象不仅可以实现抽象类中的接口方法，而且还可以调用在抽象类中定义的非纯虚函数，比如这里的 `runVirtualFunc` 方法。另外一个值得注意的地方是，在 `JavaScript` 代码中，我们也可以直接通过对应类结构的“`prototype`”原型链来调用定义在该类内部的静态方法。

6.2.8 原始指针、空指针与 void 指针

本节我们将介绍如何在 `WebIDL` 中使用原始指针、空指针和 `void` 指针。其中空指针主要是指值为“`NULL`”或“`nullptr`”的指针。

原始指针

对于原始指针在 `WebIDL` 语法中的具体表示形式，这里不再多讲。通常来说，在 `C/C++` 代码中，无论是以引用还是指针形式返回的对象结构，在对应的上层 `JavaScript` 环境中，`Emscripten` 都会帮助我们根据这些引用或指针来进行相应的类型转换，进而能够直接引用两者所对应的实体对象。那么，为了能够在 `JavaScript` 环境中直接与这些底层的指针本身进行交互，`Emscripten` 便为我们提供了多种方法用于此目的。比如 `getPointer` 方法可以获取一个指向某对象实体的整数指针值；而 `wrapPointer` 方法则与之相反，通过该方法，我们可以根据一个指针来获取其指向的实体对象。一个简单的应用示例如下：

```
webidl_pointers.cc
class WebIDL {
public:
    WebIDL (int x) : x(x) {}
    int getValueXByValue (void) const {
        return x;
    }
private:
    int x;
};
```

这里定义了一个十分简单的 `C++` 类结构，其所对应的 `WebIDL` 描述代码如下所：

```
webidl_pointers.idl
interface WebIDL {
    void WebIDL(long x);
    long getValueXByValue();
};
```

下面我们将关注点放在上层的 JavaScript 脚本代码上。

```
post-script.js
__ATPOSTRUN__.push(() => {
  // 创建一个 WebIDL 类对象
  var _t = new WebIDL(10);
  // 获得类 WebIDL 的原型
  var __class__ = _t['__class__'];
  // 获取该对象的指针
  var _t_ptr = Module['getPointer'](_t);

  console.log(_t_ptr); // 5246648;
  // 根据指针和类原型重新获取该类对象
  console.log(Module['wrapPointer'](_t_ptr, __class__)); // WebIDL {ptr: 5246648}
  Module.destroy(_t);
});
```

在这段 JavaScript 代码中，我们给出了 `getPointer` 和 `wrapPointer` 两个方法的使用方式。其中 `getPointer` 方法接收一个从 C++ 代码经过关系绑定映射过来的 JavaScript 类对象结构，并返回该对象实体的所在地址（指针值）；而 `wrapPointer` 方法则与之相反，该方法接收一个指针值及类对象的“__class__”原型，并返回一个存在于该指针位置且符合类原型的 JavaScript 类对象实体。除这两个方法外，Emscripten 还提供了其他多种可用于指针操作的上层 JavaScript 方法，读者可以参考官方文档来做进一步了解，限于篇幅这里不再介绍。

空指针

需要注意的是，从 WebIDL 函数中返回的值为“`nullptr`”或“`NULL`”的指针，并不会在上层 JavaScript 环境中被自动转换为“`null`”对象或值为“`0`”的单一数字值。相应的，该值会被转换为一个以函数返回值类型为基本类型、对应该类型对象的“`ptr`”属性值为“`0`”的包装对象。示例代码如下：

```
webidl_null_pointer.cc
class WebIDL {
public:
  WebIDL() = default;
  WebIDL* returnNullPtr() const {
    return nullptr;
  }
};
```

上述 C/C++代码对应的 WebIDL 描述文件代码如下：

```
webidl_null_pointer.idl
interface WebIDL {
    void WebIDL(long x);
    WebIDL returnNullPtr();
};
```

最后通过如下 JavaScript 代码来执行上述经过关系绑定的 returnNullPtr 函数。该函数在调用时会直接返回一个 nullptr 类型的空指针。

```
post-script.js
__ATPOSTRUN__.push(() => {
    var _t = new WebIDL(10);
    // 调用方法并返回一个空指针
    var _ptr = _t['returnNullPtr']();
    console.log(_ptr); // WebIDL {ptr: 0}
    Module.destroy(_t);
});
```

void 指针

在 WebIDL 中，我们可以使用 “VoidPtr” 或 “any” 关键字来表示一个 void 类型的普通指针。两者的区别在于：使用 “VoidPtr” 标识的指针返回值在上层 JavaScript 代码中会以 “VoidPtr 类对象” 的封装形式来表示；而使用 “any” 标识的指针则会直接以 “指针值” 的形式来表示。示例代码如下：

```
webidl_pointers.cc
class WebIDL {
public:
    WebIDL (int x) : x(x) {}
    // 两个方法均返回 void 指针
    void* returnVoidPtr (WebIDL* t) const {
        return t;
    }
    void* returnAnyPtr (WebIDL* t) const {
        return t;
    }
private:
    int x;
};
```

在上面的 C++ 代码中，我们在 WebIDL 类中定义了 `returnVoidPtr` 和 `returnAnyPtr` 两个方法，这两个方法均接收一个指向 WebIDL 对象的指针，并直接按照 `void` 类型指针再将它们返回。而在对应的 WebIDL 描述文件中，我们将分别使用 “VoidPtr” 和 “any” 关键字来标识这两个方法返回的 `void` 指针。对应的 WebIDL 描述文件代码如下：

```
webidl_pointers.idl
```

```
interface WebIDL {
    void WebIDL(long x);
    VoidPtr returnVoidPtr(WebIDL t);
    any returnAnyPtr(WebIDL t);
};
```

可以看到，当在上层 JavaScript 环境中直接调用这两个方法时，它们的返回值分别为 `VoidPtr` 类对象，以及单纯的指针值。不仅如此，我们还可以在这里模拟类似于 C/C++ 中 `void` 指针和其他类型指针之间的相互转型（cast）过程。这里需要使用 Emscripten 为我们提供的 `castObject` 和 `wrapPointer` 两个上层 JavaScript 方法。其中 `castObject` 方法主要用于多个类对象之间的转型；而 `wrapPointer` 方法则用于从某个具体的指针值来提取属于某个类型的 JavaScript 类对象。

```
post-script.js
```

```
__ATPOSTRUN__.push(() => {
    var _t = new WebIDL(10);
    // 获得类对象原型
    var __class__ = _t['__class__'];
    // 分别调用两个返回 void 指针的方法
    var voidPtrObj = _t['returnVoidPtr'](_t); // VoidPtr {ptr: 5246648}
    var voidPtrNum = _t['returnAnyPtr'](_t); // 5246648

    // 转型到 WebIDL 类对象: WebIDL {ptr: 5246648}
    console.log(Module['castObject'](voidPtrObj, __class__));
    // 通过指针值来获取该对象
    console.log(Module['wrapPointer'](voidPtrNum, __class__));
    Module.destroy(_t);
});
```

6.2.9 默认类型转换

与 Embind 类似，对于常用的 C/C++ 基本数据类型，WebIDL 会自动根据相应的关系绑定类型来向对应的 JavaScript 语法元素进行转换。并且这些转换过程均是隐式发生的，并不需要我們进行任何显式操作。示例代码如下：

```
webidl_types.cc
```

```
class WebIDL {
public:
    WebIDL () = default;
    // 返回布尔值
    bool returnBool () {
        return true;
    }
    // 返回单精度浮点数
    float returnFloat () {
        return 1.5;
    }
    // 返回双精度浮点数
    double returnDouble () {
        return 1.5;
    }
    // 返回字符值
    char returnChar () {
        return 'A';
    }
    // 返回字符指针
    const char* returnString () {
        return "WebAssembly";
    }
    // 返回无符号字符值
    unsigned char returnUChar () {
        return 'B';
    }
    // 返回无符号短整型值
    unsigned short int returnUSInt () {
        return 10;
    }
    // 返回无符号短整型值
    unsigned short returnUShort () {
        return 10;
    }
    // 返回长整型值
    unsigned long returnLong () {
        return 101;
    }
};
```

这里我们编写了多个用于测试不同基本数据类型对应上层 JavaScript 环境中其映射值的 C/C++ 方法。这些方法所对应的 WebIDL 描述代码如下：

```
webidl_types.idl
```

```
interface WebIDL {
    void WebIDL();
    boolean returnBool();
    float returnFloat();
    double returnDouble();
    byte returnChar();
    [Const] DOMString returnString();
    octet returnUChar();
    unsigned short returnUSInt();
    unsigned short returnUShort();
    unsigned long returnLong();
};
```

需要注意的是，C/C++ 语言中字符类型 “char” 对应的 WebIDL 类型标识符为 “byte”；由字符指针代表的字符串结构对应的 WebIDL 类型标识符为 “DOMString”；其他类型可以参考上述 WebIDL 文件中的代码。接下来，我们可以通过下面的 JavaScript 脚本代码来调用定义在 C/C++ 代码中的方法，并同时查看这些方法的返回值结果。

```
post-script.js
```

```
__ATPOSTRUN__.push(() => {
    var _t = new WebIDL(10);
    console.log(_t['returnBool']()); // true
    console.log(_t['returnFloat']()); // 1.5
    console.log(_t['returnDouble']()); // 1.5
    console.log(_t['returnChar']()); // 65 (A)
    console.log(_t['returnString']()); // WebAssembly
    console.log(_t['returnUChar']()); // 66 (B)
    console.log(_t['returnUSInt']()); // 10
    console.log(_t['returnUShort']()); // 10
    console.log(_t['returnLong']()); // 10
    Module.destroy(_t);
});
```

第 7 章

探索 Emscripten 高级特性

提示：本章使用的 emsdk 工具包是 1.38.0 版本。

通过第 6 章的介绍，我们已经了解如何基于 Emscripten 来实现 C/C++ 和 JavaScript 语言在语法元素上的关系绑定过程。在本章中，我们将继续探索 Emscripten 的高级特性，这些特性将支持我们在 Wasm 应用中使用传统 Linux 文件系统 API、浏览器的上层事件系统，以及各类与多媒体（音/视频）编程相关的常用组件库。这无疑会使 Wasm 应用的功能变得更加丰富，同时也使得现有的 C/C++ 应用向 Web 平台（WebAssembly/ASM.js）的移植过程变得更加简单和无痛。

7.1 加入优化流程

当在浏览器中运行一个完整的 WebAssembly 应用时，需要经历：从远程位置下载模块二进制文件、下载用于连接上层 JavaScript 环境与模块实例的“胶水”脚本文件、初始化模块需要使用的内存和变量资源等环节。而一旦其中某个环节消耗的时间过长，WebAssembly 本身所带来的性能优势便无法得到完全体现，甚至应用的整体运行效率可能还会低于具有同样功能但完全基于 JavaScript 语言编写的应用。因此，在本节中，我们将首先讨论如何从 Wasm 应用的各个环节来优化其整体性能。

首先，我们会构建一个简单的 Wasm 应用，然后以该应用作为示例来进一步介绍 Emscripten 提供的一系列可用的优化流程。借助这些优化流程，我们可以从减小生成模块的文件体积、模块对象的加载与重用以及“胶水”脚本文件的代码压缩等多个方面来优化并提高 Wasm 应用的整体运行效率。下面直接给出该应用的 C/C++ 代码。

```
QuickSort.cc
#include <iostream>
```



```
#include <vector>
#include <emscripten.h>

using namespace std;

// 定义用于交换两个位置元素值的函数
void swap (char* a, char* b) {
    char t = *a;
    *a = *b;
    *b = t;
}

// 打印数组内容
void printArr (char arr[], char length) {
    vector<char> t(arr, arr + length);
    for (auto &e : t) {
        cout << (int)e << " ";
    }
    cout << endl;
}

// 函数以序列最右侧的元素作为“基准数”进行交换过程
char partition (char arr[], char low, char high) {
    // 选择基准数
    char pivot = arr[high];
    char i = (low - 1);

    // 遍历元素，并交换基准数两侧的元素
    for (char j = low; j <= high - 1; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    // 置位基准数
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
```

```
// 模块向 JavaScript 环境暴露的主要方法
extern "C" char* EMSCRIPTEN_KEEPALIVE quickSort(char arr[], char low, char high) {
    if (low < high) {
        printArr(arr, high + 1);
        // 获得下一次交换的序列区间
        char pi = partition(arr, low, high);

        // 继续递归排列左、右两部分的子序列
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
    return arr;
}
```

上述 C/C++代码相对复杂一些，其主要功能是对 JavaScript 环境中的任意整数数组，通过快速排序算法对其内部的元素进行升序排列。关于快速排序算法的内容这里不做介绍，读者可以根据代码中的注释信息进行理解。在这段 C/C++代码中，我们将用于上层 JavaScript 脚本的 `quickSort` 方法通过 `EMSCRIPTEN_KEEPALIVE` 标识导出，该方法一共接收三个参数：待排序数组的首地址、待排序序列的最低位索引值和最高位索引值（区间）。当该 Wasm 应用在运行时，还会通过 `printArr` 方法将算法每一次递归时传递给 `quickSort` 方法的子序列内容打印出来。接下来，我们编写用于调用该模块方法的上层 JavaScript 脚本代码。

```
post-script.js
```

```
__ATPOSTRUN__.push(() => {
    // 定义一个用于快速排序的数组
    var array = [11, 24, 36, 42, 4, 25, 9, 1, 0];
    // 调用模块中的方法，并返回该数组的首地址
    var arrayPointer = Module['cwrap'](
        'quickSort', 'number', ['array', 'number', 'number']
    )(array, 0, array.length - 1);
    // 定义一个结果集容器
    var clearArrResult = [];
    for (let i = 0; i < array.length; i++) {
        // 通过 Emscripten 运行时环境提供的 Module.getValue 方法从共享线性内存中提取数据
        clearArrResult.push(Module['getValue'](arrayPointer + i, 'i8'));
    }
    console.log(clearArrResult);
});
```

这里直接通过 `Module.cwrap` 方法来封装并间接调用从模块中导出的 `quickSort` 方法。但需

要注意的是，对于 `array` 类型的函数实参，`Module.cwrap` 方法在其内部是通过“HEAP8”内存模型访问符在共享线性内存中为其分配数据的，因此，实际上这些数据是按照 8 位长整数的格式，通过 JavaScript 中的 `Int8Array` 类型数组被存储到模块的共享线性内存中的。所以，当模块通过 C/C++ 代码从内存中取出数据时，也需要在代码中使用与 JavaScript 侧长度一致的数据类型。这里可以看到，在 C/C++ 代码中，我们将传递给 `quickSort` 方法的数组声明为“char”类型，即字符数组。而这正好与存放在共享线性内存中的数组元素其数据长度相符合。代码编写完成后，可以通过如下命令来编译该 Wasm 应用。

```
emcc QuickSort.cc
--std=c++11
-s WASM=1
-o QuickSort.html
--post-js post-script.js
-s EXTRA_EXPORTED_RUNTIME_METHODS="['cwrap', 'getValue']"
```

接下来运行该应用，即可在浏览器的 Console（控制台）中看到如图 7-1 所示的运行结果。

09:13:49.238 11 24 36 42 4 25 9 1 0	QuickSort.html:1237
09:13:49.240 0 24 36 42 4 25 9 1 11	QuickSort.html:1237
09:13:49.241 0 4 9 1	QuickSort.html:1237
09:13:49.242 0 1 9 4	QuickSort.html:1237
09:13:49.243 0 1 4 9 11 25 36 42 24	QuickSort.html:1237
09:13:49.244 0 1 4 9 11 24 36 42 25	QuickSort.html:1237
09:13:49.246 0 1 4 9 11 24 25 42 36	QuickSort.html:1237
09:13:49.248 ▶ (9) [0, 1, 4, 9, 11, 24, 25, 36, 42]	QuickSort.js:6426

图7-1 上述Wasm应用在浏览器中的运行结果

现在，我们来为该应用添加与性能测试相关的 JavaScript 代码。这里主要通过浏览器提供的 `Performance` 相关接口来记录应用的整体运行时间、加载时间和脚本执行时间。整体运行时间是指页面从开始加载到 JavaScript 脚本完全执行完毕，即向控制台输出数组的最终排序结果时所花费的时间；加载时间是指从网页开始加载到 `load` 方法被执行，即 DOM 和各资源文件已经下载并准备完毕时所用的时间；脚本执行时间则以 Emscripten “胶水”脚本文件内部代码的执行时间为准，为了记录这段时间，我们需要在该文件的第一行代码中记录下当前时刻脚本执行的起始时间，这里将通过 `emcc` 编译器的“`--pre-js`”参数来实现这个需求。如下所示，我们将所有希望拼接到“胶水”脚本文件头部的 JavaScript 代码写到名为“`pre-script.js`”的脚本文件中。

```
pre-script.js
var startTimestamp = performance.now();
```

可以看到，上面这行代码将浏览器在执行到当前 JavaScript 代码时的时间点信息存储到了名为“`startTimestamp`”的全局变量中，该变量将会在后端的代码中被使用。接下来，我们需要

改写 `post-script.js` 脚本文件的内容，将如下这段用于性能测试的代码追加到该文件最后。注意：要将代码整体放到 “`__ATPOSTRUN__`” 所对应的生命周期钩子队列中。

```
post-script.js
```

```
...
// 性能测试记录
var time = performance.timing;
var loadingTime = time.loadEventStart - time.navigationStart;
var executionTime = performance.now() - startTimestamp;
var totalTime = new Date().getTime() - time.navigationStart;
// 打印应用的脚本执行时间
console.log("Application ET: ", Math.round(executionTime), "ms");
// 打印应用的脚本下载时间和初始化时间
console.log("Application LT: ", loadingTime, "ms");
// 打印应用的整体运行时间
console.log("Application TT: ", totalTime, "ms");
```

在加入了用于进行性能测试的 JavaScript 脚本代码段后，我们可以使用如下命令来重新编译该 Wasm 应用。

```
emcc QuickSort.cc
--std=c++11
-s WASM=1
-o QuickSort.html
--post-js post-script.js
--pre-js pre-script.js
-s EXTRA_EXPORTED_RUNTIME_METHODS=['cwrap', 'getValue']"
```

我们重新加载并运行该应用，可以看到在浏览器的 Console（控制台）中多输出了如下三条信息，如图 7-2 所示。其中第一条信息表示整个 Wasm 应用的脚本执行（JavaScript 代码执行 + Wasm 模块导出方法的调用）时间；第二条信息表示该 Wasm 应用的相关资源在进行远程加载时所消耗的时间；第三条信息表示整个 Wasm 应用从浏览器页面打开到运行完毕，即输出结果所花费的时间。这些时间均以“毫秒”为单位。由于 JavaScript 代码可能在页面的 load 事件被调用前就开始执行，因此，在 ET 和 LT 阶段的耗时之和并不与 TT 阶段相等。

23:19:36.253 Application ET: 469 ms	QuickSort.js:6437
23:19:36.253 Application LT: 533 ms	QuickSort.js:6438
23:19:36.254 Application TT: 938 ms	QuickSort.js:6439

图7-2 上述Wasm应用在运行时输出的性能测试信息

现在，我们记录下当前 Wasm 应用所加载相关静态资源文件的大小。通过浏览器的“Network

(网络)”选项卡可以看到，从远程加载的 JavaScript “胶水”脚本文件其大小为 269KB，而 Wasm 模块二进制文件的大小为 323KB，它们都是在对当前 Wasm 应用没有进行任何优化策略时得到的数据。接下来，我们将为该 Wasm 应用逐渐增加 Emscripten 提供的甚至没有提供的各种优化流程，并从 ET、LT 和 TT 这三个阶段所消耗的时间长短来观察这些优化流程的实际效果。

7.1.1 使用编译器代码优化策略

前面介绍过，整个 Emscripten 编译器链路的前端部分是基于 Clang 的编译器前端进行构建的，这意味着我们可以在 emcc 命令中直接使用 Clang 支持的所有优化参数。这些优化参数可以在一定程度上影响整个 WebAssembly 应用从 C/C++ 源代码的编译到 LLVM-IR 代码的生成过程。不仅如此，基于 Emscripten 的内部封装，其编译器链路的入口脚本 emcc，也会根据这些传递给 Clang 的优化参数来分别调用 Binaryen 工具链及内部的 JavaScript 代码优化器，对 Wasm 模块本身及“胶水”脚本文件的内部代码进行优化。

通过 emcc 命令，我们可以在编译 Wasm 应用时使用如下几种代码优化参数。每一个参数的优化级别随着其最后一位数字的增大而升高，较高的优化级别会逐渐引入更多的代码优化策略，但同时也会相应增加应用的编译时间。

-O0

“-O0”参数不会对当前应用的代码进行任何优化。

-O1

“-O1”参数仅会对应用代码进行轻微程度的优化，其优化内容主要包括：删除“胶水”脚本文件中的断言测试代码，以及 C++ 代码中的异常捕获代码，并开启“ALIASING_FUNCTION_POINTERS”选项，以启用函数指针别名。该优化参数主要侧重于构建一个将会花费较短时间的编译流程，用以缩短应用测试时的编译迭代周期。

-O2

“-O2”参数会在“-O1”参数的基础上对代码做进一步的优化。编译器会通过调用内部的 JavaScript 代码优化器来移除部分没有使用到的脚本代码（包括通过“--pre-js”和“--post-js”参数拼接到“胶水”脚本文件中的代码）。该等级的优化参数可以进一步减小整个 Wasm 应用对应代码资源所占用的体积，但可能并不适合使用在即将发布的应用上。

-O3

“-O3”参数会在“-O2”参数的基础上对代码做更进一步的优化。该优化参数不会过度地

对 Wasm 二进制模块本身，以及“胶水”脚本文件中的 JavaScript 代码进行破坏性的优化，但又在最大程度上精简了代码的内容，让整个应用的加载和运行效率达到最优。因此，该等级的优化参数更适合使用在即将发布到生产环境的 Wasm 应用上。

-Os

“-Os”参数与“-O3”参数类似，但“-Os”更侧重于优化整个 Wasm 应用的代码体积，因此它可能会牺牲一部分应用的运行性能。该优化参数将会调用与 LLVM 相关的优化器及其内置的 JavaScript 代码优化器，来对整个应用代码及相关依赖进行更深层的代码组成优化。

-Oz

“-Oz”参数与“-Os”参数类似，但是它会更进一步地优化 Wasm 应用的整体代码体积。但需要注意的是，由于其过度的代码优化过程，可能会导致应用出现不可预见的运行时错误。

对于上面列出的一系列优化参数，我们给出的使用建议是：请在首次编译应用代码时，尝试使用默认参数（即不进行任何优化）来进行编译过程。当确保在默认参数下 Wasm 应用可以正常运行时，再尝试使用更高级别的优化参数来优化应用代码。这是由于优化器可能对代码的组成结构和具体写法有着严格的要求。因此，请在确保应用本身的组成代码没有明显语法错误和业务逻辑错误的情况下，再增加另外的优化流程。而这样做将会让应用本身的调试过程变得更加简单。

接下来，我们会重新编译之前的示例应用，并通过增加“-O3”参数来为整个 Wasm 应用加入代码优化流程。修改后的编译命令语句如下：

```
emcc QuickSort.cc
-O3
--std=c++11
-s WASM=1
-o QuickSort.html
--post-js post-script.js
--pre-js pre-script.js
-s EXTRA_EXPORTED_RUNTIME_METHODS=['cwrap', 'getValue']"
```

当编译完成后，在浏览器中运行该应用，并同时观察整个应用在运行时输出的性能测试数据。如图 7-3 示为在多次运行该应用后我们选取的中间测试结果。

可以看到，该应用在经过“-O3”级别的代码优化处理后，其脚本运行时间（ET）和应用整体运行时间（TT）均有明显的下降，而与应用相关资源的远程加载时间（LT）也有小幅度的

下降。此时我们再观察浏览器的“Network（网络）”选项卡，会发现整个 Wasm 应用所依赖的 JavaScript “胶水”脚本文件其大小为 110KB，对应的 Wasm 模块二进制文件大小为 161KB。在下一节的内容中，我们将通过 GCC 来进一步压缩“胶水”脚本文件内的 JavaScript 代码，进而间接地降低应用在 LT（资源加载）阶段所消耗的时间。

23:18:36.251 11 24 36 42 4 25 9 1 0	QuickSort.html:1237
23:18:36.253 0 24 36 42 4 25 9 1 11	QuickSort.html:1237
23:18:36.254 0 4 9 1	QuickSort.html:1237
23:18:36.256 0 1 9 4	QuickSort.html:1237
23:18:36.257 0 1 4 9 11 25 36 42 24	QuickSort.html:1237
23:18:36.259 0 1 4 9 11 24 36 42 25	QuickSort.html:1237
23:18:36.260 0 1 4 9 11 24 25 42 36	QuickSort.html:1237
23:18:36.261 ▶ (9) [0, 1, 4, 9, 11, 24, 25, 36, 42]	QuickSort.js:1
23:18:36.264 Application ET: 324 ms	QuickSort.js:1
23:18:36.264 Application LT: 504 ms	QuickSort.js:1
23:18:36.265 Application TT: 779 ms	QuickSort.js:1

图7-3 加入“编译器优化”后应用运行时输出的性能测试信息

7.1.2 使用 GCC 压缩代码

这里要介绍的 GCC 并不是大家熟知的 GUN 编译器集合，而是由 Google 开发的一款专门用于压缩 JavaScript 脚本代码的优化器，其英文全称为“Google Closure Compiler”。借助该优化器的强大特性，我们可以最大程度地压缩 Emscripten 在编译 Wasm 应用时生成的“胶水”脚本文件中的 JavaScript 代码。

由于 GCC 本身是基于 Java 语言编写的，因此在使用该优化器前，请确认本机已经预装了 JRE（Java 运行时环境），并确保可以在命令行的全局环境下直接调用“java”命令。相比于我们常用的“UglifyJS”压缩库，GCC 在 JavaScript 代码的压缩优化策略上会显得更加激进，在某种程度上可能会具有一定的破坏性，即在压缩过程中破坏了原始代码的可读性结构。但也正是由于这种对源代码的“分析破坏”和“再重构”过程，使得 GCC 成为压缩率最高的 JavaScript 代码优化工具。

如果想要使用 GCC 对当前的 Wasm 应用进行优化，则需要对上层的 JavaScript 脚本代码做出一些改变。如果你仔细观察就会发现，在调用 Emscripten 运行时环境所提供的 `cwrap` 方法时，我们是通过“`Module[cwrap]`”这种字面量属性值的方式来调用的。而这就是专门为使用 GCC 而做出的改变。实际上，在使用 GCC 压缩代码时，它会将所有“`Module.cwrap`”这种属性值形式的函数调用过程进行重构，并压缩属性名的字符串长度。而以字面量属性值形式表示的调用过程则会被“原封不动”地保留下来。因此，这里需要与 Emscripten “胶水”脚本内部的函数

定义和调用方式保持一致，即同时以字面量属性值的形式来进行函数的定义和调用过程。

在确保上层 JavaScript 代码的语法形式符合 GCC 的优化条件后，我们便可以通过在编译命令语句中添加“`--closure 1`”参数，来让 Emscripten 在进行代码优化过程时自动调用 GCC 优化“胶水”脚本文件中的 JavaScript 代码。

```
emcc QuickSort.cc
-O3
--closure 1
--std=c++11
-s WASM=1
-o QuickSort.html
--post-js post-script.js
--pre-js pre-script.js
-s EXTRA_EXPORTED_RUNTIME_METHODS="['cwrap', 'getValue']"
```

当应用编译完成后，我们通过浏览器来运行该应用，可以在 Console 中看到如图 7-4 所示的性能测试信息（这里仍然是多次运行该应用并选取了中间结果）。

00:32:09.091 11 24 36 42 4 25 9 1 0	QuickSort.html:1237
00:32:09.095 0 24 36 42 4 25 9 1 11	QuickSort.html:1237
00:32:09.097 0 4 9 1	QuickSort.html:1237
00:32:09.098 0 1 9 4	QuickSort.html:1237
00:32:09.100 0 1 4 9 11 25 36 42 24	QuickSort.html:1237
00:32:09.101 0 1 4 9 11 24 36 42 25	QuickSort.html:1237
00:32:09.103 0 1 4 9 11 24 25 42 36	QuickSort.html:1237
00:32:09.105 ► (9) [0, 1, 4, 9, 11, 24, 25, 36, 42]	QuickSort.js:94
00:32:09.110 Application ET: 371 ms	QuickSort.js:94
00:32:09.110 Application LT: 431 ms	QuickSort.js:94
00:32:09.111 Application TT: 709 ms	QuickSort.js:94

图7-4 加入“编译器优化”和“GCC优化”后应用在运行时输出的性能测试信息

可以看到，经过 GCC 的优化后，整个 Wasm 应用在 LT 阶段的资源加载时间有了小幅度的下降。此时我们通过浏览器的“Network”选项卡可以看到，当前 Wasm 应用加载的 JavaScript “胶水”脚本文件其大小只有 41.9KB，而 Wasm 模块二进制文件的大小仍然是 161KB。由此可见，GCC 对 JavaScript 代码的优化效果非常显著。

最后需要强调的是，GCC 还可以通过使用源代码内关于各 JavaScript 变量的数据类型信息来增强代码的实际优化效果。而对于这些变量的类型信息则需要由 JSDoc 文档生成工具所使用的注释段来进行描述，GCC 在运行时会在 JSDoc 的注释标签中查找变量的类型信息，然后根据这些信息来对代码做更进一步的优化。关于 GCC 更详细的使用方法可以参考 Github 上的官方

文档，限于篇幅这里不再展开介绍。

7.1.3 使用 IndexedDB 缓存模块对象

通过使用前面介绍的几种优化流程进行优化后，Wasm 应用的 JavaScript 脚本文件其体积已经被压缩到一个理想的大小，如果再配合服务器端的 Gzip 等压缩算法，文件的体积可以进一步减小。因此，本节我们将把重点放在 Wasm 模块的二进制文件上，即如何才能进一步减小该文件的体积，或者提高浏览器对 Wasm 模块的加载和初始化速度。

实际上，我们可以通过使用“-Oz”级别的优化参数来进一步压缩 Wasm 模块二进制文件的体积，但这通常会在一定程度上牺牲应用整体的运行效率。因此，本节我们将尝试通过另一种方式来间接地提高浏览器加载和初始化 Wasm 模块的速度。

按照正常的 Wasm 应用加载和运行流程，从页面载入时开始计算，需要经历如下几个阶段：①加载应用所需要的 JavaScript 脚本及 Wasm 模块二进制文件；②通过 `WebAssembly.instantiate()` 等方法实例化一个 Wasm 对象；③通过该对象来间接地调用模块暴露出的方法。因此，从另一个角度来看，既然模块文件本身的体积很难再进行优化，那么是否可以直接省略从远程服务器加载模块文件的过程，进而间接提升整个应用的运行效率呢？答案是：对于非首次运行的 Wasm 应用来说是可以的。不仅如此，还可以同时省略掉模块的部分初始化过程，让从第二次开始运行的 Wasm 应用可以直接获取到模块对应的 `WebAssembly.Module` 对象。而整个方案便是基于浏览器提供的可以进行对象数据缓存的 IndexedDB 数据库来实现的。

相比于 HTML 1.1，HTML 5 的一个最重要的特性就是提供了可以用于本地数据持久化的相关功能，这使得用户可以在线或离线地访问 Web 应用。对于简单的键值对数据，我们可以通过 `localStorage` 或 `sessionStorage` 对其进行本地的持久型/会话型数据存储；而对于类似 `WebAssembly.Module` 这类复杂的对象数据，便需要借助浏览器强大的 IndexedDB 数据库来进行存储。

在如图 7-5 所示的 WebAssembly 上层 Web 接口标准文档中，我们可以看到 `WebAssembly.Module` 被定义成一个具有“可克隆”属性的对象。这就意味着我们可以将该对象直接存储在浏览器的本地 IndexedDB 数据库中，并可以随时随地对其进行对象的再提取、数据更新与删除等操作。同样的，我们也可以将这些 Module 对象通过 `Worker.postMessage` 方法在多个 Worker 线程中任意进行传递和使用。而这些都是“可克隆”对象所独有的特性。

为了能够在基于 Emscripten 生成的 WebAssembly 应用中使用这些特性，我们需要首先完成并注意以下几件事情。从整体上看，这些事情可能会稍显复杂，但是面对基于 IndexedDB 所带

来的 Wasm 应用性能提升，完成这些事情所花费的精力就显得微不足道了。

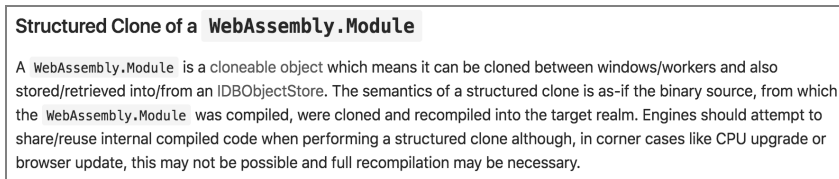


图7-5 标准文档中对WebAssembly.Module对象“可克隆”属性的说明

(1) 由于我们在上述 Wasm 应用的 C/C++源代码中使用了一些需要依赖于 Emscripten 运行时环境实现的 C++特性（如 `cout` 对象对应的控制台输出功能），因此，这里无法将整个项目直接编译为 Standalone 类型的 Wasm 应用（无论是通过 Emscripten 优化器还是将模块编译成动态库，在 C/C++源代码中使用的特殊方法都需要由外部的 JavaScript 函数来实现）。而为了使应用的性能测试能够基于统一的模块数据来进行，因此需要直接对 Emscripten 生成的“胶水”脚本文件进行修改，以在其内部添加对模块对象的 IndexedDB 缓存支持。

(2) IndexedDB 能够存储 WebAssembly.Module 对象，依赖浏览器对该对象进行的特殊底层处理。但这种存储方式的实现本身较为复杂，并且也存在诸多问题，比如使用 IndexedDB 在一个不安全的 JavaScript 上下文中缓存模块对象，可能会造成来自非法域的中间人攻击（MITM）。另外，使用 IndexedDB 来缓存 Wasm 模块，使得 IndexedDB 失去了其本身更为重要的应用场景。相对于重度依赖 IndexedDB 的本地存储功能，一种更好的解决办法就是为 Wasm 模块提供专用的缓存机制。由于上面提到的一系列原因，最新版本的 Chrome 浏览器已经默认关闭了其内部 IndexedDB 数据库对 WebAssembly.Module 对象的存储支持。因此，如果想要在 Chrome 浏览器中使用该特性，则需要通过“chrome://flags/”进入 Chrome 的实验选项列表中，并开启名为“WebAssembly 结构化克隆支持”的选项，如图 7-6 所示。

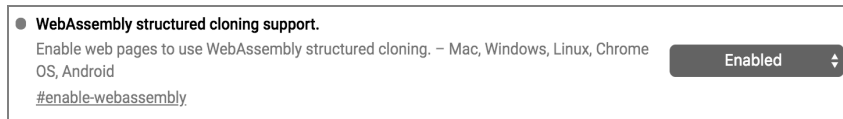


图7-6 开启Chrome浏览器中的“WebAssembly结构化克隆支持”选项

为了能够让 Emscripten 在编译项目中支持对 Wasm 模块的 IndexedDB 缓存，笔者直接为该项目源代码提交了新的“Pull Request”用以支持该功能。这里可以直接在编译命令中添加名为“MODULE_CACHE”的参数来启用本地 IndexedDB 数据库对 Wasm 模块对象的缓存功能。修改后的编译命令如下：

```
emcc QuickSort.cc
--std=c++11
-s MODULE_CACHE="[1, 'QuickSort']"
-s WASM=1
-o QuickSort.html
--post-js post-script.js
--pre-js pre-script.js
-s EXTRA_EXPORTED_RUNTIME_METHODS=['cwrap', 'getValue']"
```

这里的“MODULE_CACHE”参数需要接收一个数组字面值形式的字符串。在该数组中共包含两个字段，分别为当前应用的“整数版本号”和“应用名称”。这些字段将会在应用编译完成后分别用来作为 IndexedDB 的数据库版本号和对应的数据项键名。接下来，我们再次通过浏览器来运行该应用，并观察其在控制台中输出的性能测试信息。

如图 7-7 所示为第二次运行该 Wasm 应用时，其打印出的性能测试数据。可以很明显地看到，该应用在 ET 阶段的耗时有了一定程度的下降（371ms->190ms），而所减少的时间正好是之前“胶水”脚本通过 WebAssembly.instantiateStreaming 接口从远程拉取 Wasm 模块所消耗的时间。不仅如此，此时如果在浏览器的“Network”选项卡中查看应用静态资源的加载情况，便可发现其只加载了一个大小为 43.5KB 的 JavaScript“胶水”脚本文件。

14:45:05.333 Found module in wasm cache: QuickSort	QuickSort.js:10
14:45:05.395 11 24 36 42 4 25 9 1 0	QuickSort.html:1237
14:45:05.398 0 24 36 42 4 25 9 1 11	QuickSort.html:1237
14:45:05.399 0 4 9 1	QuickSort.html:1237
14:45:05.400 0 1 9 4	QuickSort.html:1237
14:45:05.401 0 1 4 9 11 25 36 42 24	QuickSort.html:1237
14:45:05.403 0 1 4 9 11 24 36 42 25	QuickSort.html:1237
14:45:05.404 0 1 4 9 11 24 25 42 36	QuickSort.html:1237
14:45:05.406 ► (9) [0, 1, 4, 9, 11, 24, 25, 36, 42]	QuickSort.js:97
14:45:05.412 Application ET: 190 ms	QuickSort.js:97
14:45:05.413 Application LT: 496 ms	QuickSort.js:97
14:45:05.413 Application TT: 591 ms	QuickSort.js:97

图7-7 使用了IndexedDB模块对象缓存功能后应用输出的性能测试信息

现在切换到浏览器开发者工具的“Application”选项卡，通过左侧的“IndexedDB”项可以查看到当前浏览器 IndexedDB 数据库内的数据存储情况。如图 7-8 所示，可以看到，在名为“\$\$_wasm_module_instances”的对象表中，当前应用所对应的 WebAssembly.Module 模块对象以“键值对”的形式被存储在这里。因此，在确保模块编译时通过“MODULE_CACHE”参数指定的“数据库版本号”和“应用名称”均不变的情况下，Wasm 应用从其第二次开始运行时起将直接使用存储在 IndexedDB 中的模块对象来进行模块实例化。

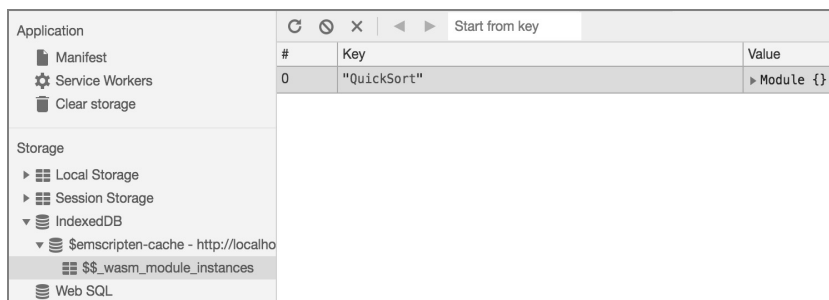


图7-8 查看缓存在IndexedDB数据库中的WebAssembly.Module对象

如下所示为 Emscripten “胶水” 脚本文件在其内部调用的 `cacheModuleInstance` 方法的代码实现细节。该方法将会根据所传入参数个数的不同来区分模块对象的存储和提取过程。从上层编译命令传入 `emcc` 编译器入口脚本的“MODULE_CACHE”参数，将会由编译器链路内部的预处理过程进行处理，处理后的参数值将会以逗号分隔值（CSV）的形式被替换到“胶水”脚本文件的对应位置上（比如这里 `MODULE_CACHE_PARAMS` 变量对应的赋值单引号中）。关于 `cacheModuleInstance` 方法的代码实现细节，由于其较为简单这里不再展开介绍，读者可以参考注释来自行理解。

QuickSort.js

```
...
// 声明数据库版本号和应用名称等参数
var DBVERSION, APP_NAME;
// 获取从 emcc 传递过来的版本号和应用名称（由预处理过程直接替换）
var MODULE_CACHE_PARAMS = '1,QuickSort';
MODULE_CACHE_PARAMS.split(',').forEach(function(item) {
  if(!DBVERSION && /^d+$/.test(item)) {
    DBVERSION = Number(item);
  } else {
    APP_NAME = item;
  }
});
// 用于缓存模块对象的主方法
function cacheModuleInstance(appName, moduleInstance) {
  if (!indexedDB) {
    Module.printErr('Warning: Your current environment does not support IndexedDB!');
    return;
  }
  // 定义数据库名和对象表名
  var dbName = '$emscripten-cache';
```

```
var storeName = '$$_wasm_module_instances';
// 定义打开数据库的方法
function openDatabase() {
  return new Promise(function (resolve, reject) {
    var request = indexedDB.open(dbName, DBVERSION);
    request.onerror = reject.bind(null, 'Error opening wasm cache database.');
```

request.onsuccess = function() {

 resolve(request.result)

};

request.onupgradeneeded = function(event) {

 var db = request.result;

 if (db.objectStoreNames.contains(storeName)) {

 console.log('Clearing out module cache at version: ' + event.oldVersion);

 db.deleteObjectStore(storeName);

 }

 console.log('Creating wasm cache at version: ' + event.newVersion);

 db.createObjectStore(storeName)

 };

});

}

// 定义用于查询模块实体的方法（查询过程基于对象表名和应用名称）

function lookupInDatabase(db) {

 return new Promise(function (resolve, reject) {

 var store = db.transaction([storeName]).objectStore(storeName);

 var request = store.get(appName);

 request.onerror = reject.bind(null, 'Error getting wasm module' + appName);

 request.onsuccess = function(event) {

 if (request.result)

 resolve(request.result);

 else

 reject('Module was not found in wasm cache for: ' + appName);

 }

 });

}

// 定义用于存储模块对象的方法

function storeInDatabase(db, module) {

 var store = db.transaction([storeName], 'readwrite').objectStore(storeName);

```

var request = store.put(module, appName);
request.onerror = function(err) {
    console.log('Failed to store in wasm cache: ' + err)
};
request.onsuccess = function(err) {
    console.log('New record stored in wasm cache: ' + appName)
};
}

// 模块缓存的主流程
return openDatabase().then(function(db) {
    if (moduleInstance) {
        // 检查待存储数据的实际类型
        if (Object.prototype.toString.call(moduleInstance) === '[object
WebAssembly.Module]')
            storeInDatabase(db, moduleInstance);
    } else {
        // 查找模块并返回对象
        return lookupInDatabase(db).then(function(module) {
            console.log('Found module in wasm cache: ' + appName);
            return module;
        }, function(err) {
            return false;
        })
    }
});
}
...

```

关于 `cacheModuleInstance` 方法的实际调用细节可以参考图 7-9 所示的代码。这里在每一次实例化模块对象时，都会先通过该方法尝试从本地的 `IndexedDB` 数据库中来提取模块所对应的 `WebAssembly.Module` 对象，若提取不到，则再由 `WebAssembly.instantiateStreaming` 方法从原始的远程位置上进行拉取。需要注意的是，这里存储在 `IndexedDB` 数据库中的并不是已经完成实例化的 `WebAssembly.Instances` 模块对象实例，因此在继续使用前，还需要通过 `WebAssembly.instantiate` 方法并结合模块的导入信息进行模块的异步实例化。

可以看到，基于 `IndexedDB` 实现的模块对象缓存确实可以在一定程度上提升 Wasm 应用的整体运行效率，但是这种方式其本身所存在的一些问题无法被正在快速发展的 `WebAssembly` 标准所忽视。不出所料，就在笔者编写本章内容的当天晚上，`Chromium` 社区关于该话题的讨论组

便有官方人员给出了答复，如图 7-10 所示。其表达的主要内容为：浏览器仍会为 WebAssembly 提供某种更好的缓存机制，但是当前对 WebAssembly.Module 对象支持的结构化克隆特性将会被移除，即无法再通过 IndexedDB 来缓存该类型的对象。由此可见当前 WebAssembly 标准的版本迭代速度之快。

```

1686 cacheModuleInstance(APP_NAME).then(function(module) {
1   if (module) {
2     WebAssembly.instantiate(module, info).then(function(instance) {
3       receiveInstantiatedSource({
4         instance: instance,
5         module: module
6       })
7     });
8   } else {
9     // Prefer streaming instantiation if available.
10    if (!Module['wasmBinary'] &&
11        typeof WebAssembly.instantiateStreaming === 'function' &&
12        !isDataURI(wasmBinaryFile) &&
13        typeof fetch === 'function') {
14      WebAssembly.instantiateStreaming(fetch(wasmBinaryFile, { credentials: 'same-origin' }), info)
15        .then(function(output) {
16          receiveInstantiatedSource(output);
17          cacheModuleInstance(APP_NAME, output['module']);
18        })
19        .catch(function(reason) {
20          // We expect the most common failure cause to be a bad MIME type for the binary,
21          // in which case falling back to ArrayBuffer instantiation should work.
22          Module['printErr']('wasm streaming compile failed: ' + reason);
23          Module['printErr']('falling back to ArrayBuffer instantiation');
24          instantiateArrayBuffer(receiveInstantiatedSource);
25        });
26    } else {
27      instantiateArrayBuffer(receiveInstantiatedSource);
28    }
29  }
30 });

```

图7-9 cacheModuleInstance方法的实际调用细节

Comment 20 by pwnall@chromium.org, Today (15 hours ago)

Status: Won't Fix (was: Started)

Won't Fix-ing this, as WASM support in IDB is canceled.

Comment 21 by steip...@gmail.com, Today (11 hours ago)

On noes. Are there any details we can read up on the reason this has been cancelled?

Comment 22 by pwnall@chromium.org, Today (5 hours ago)

For clarity: WASM will still have a caching story. It's just that the details will involve a mechanism that is not IDB.

Using IDB to cache WASM adds a lot of unnecessary complexity, and we expect that other avenues will yield a faster path to WASM caching. You can get some ideas about the complexities involved by reading <https://docs.google.com/document/d/1PntTuoo3MKGjfPEVYA0SC01NYRdpuCMxRRjHjW3Z10A/> and <https://docs.google.com/document/d/1TgUeRIza0RC3uAzX1a5rH1b5jUHKbLjsefU--zdYaq/>

图7-10 社区决定不再支持将WebAssembly.Module对象缓存到IndexedDB数据库中

也正是由于 Chromium 的官方答复，致使笔者之前的“PR”最终没有被正式合并到 Emscripten 工具链的代码库中。但读者仍可以在笔者的“Fork”版本仓库中找到与这部分提交内容相关的

代码（commit: 0bc782b9aace3fb5f07780c08ba7f4f496680cf4）。不过，按照之前的标准与实现迭代速度，距离新缓存方案的完全实现还需要很长一段时间。而在这段时间里，我们不妨暂时还是使用 IndexedDB 来提高 WebAssembly 应用的整体加载速度，笔者后续也会将能够自动根据浏览器进行降级的 IndexedDB 缓存方案提交到“Fork”版本的 Emscripten 源代码仓库中。

7.1.4 其他优化参数

除上面介绍的几种常用优化策略外，还有如下一些可用于特定场景中的编译器优化参数。不过，在使用这些参数时，需要注意其产生的优化效果是否会影响原始应用对应 C/C++ 代码的编译过程，以及“胶水”脚本文件的正常使用。

`-s NO_FILESYSTEM=1`

通过“`-s NO_FILESYSTEM=1`”参数，可以让 `emcc` 在编译 Wasm 应用时禁用对虚拟文件系统的支持，这在某种程度上可以同时减小模块二进制文件与“胶水”脚本文件的体积。但使用该参数的前提是，没有在应用代码中使用与虚拟文件系统相关的 API 和特性。

`-fno-rtti -fno-exceptions`

通过“`-fno-rtti -fno-exceptions`”参数，我们能够关闭 `emcc` 在编译 C/C++ 源代码时可能会使用的运行时类型识别（RTTI）特性，进而在一定程度上减小所生成代码的体积。

`--llvm-lto 0/1/2/3`

通过“`--llvm-lto 0/1/2/3`”参数，我们能够在模块的编译过程中同时启用 LLVM 的链接时优化器，来对中间形式的 LLVM-IR 代码进行优化。但实际上，该优化方式存在一些已知问题，可能会对最后生成的 Wasm 应用的正常功能有所影响。因此，若需使用该优化参数，请确保对编译后生成的 Wasm 应用进行覆盖率足够高的功能性回归测试。

在本节的最后，我们将使用上面介绍的所有可用的优化手段来优化之前给出的 Wasm 应用。如下为最终的编译命令。

```
emcc QuickSort.cc
-O3
--closure 1
--std=c++11
--post-js post-script.js
--pre-js pre-script.js
--llvm-lto 3
```



```

-s AGGRESSIVE_VARIABLE_ELIMINATION=1
-s MODULE_CACHE="[1, 'QuickSort']"
-s NO_FILESYSTEM=1
-s WASM=1
-s EXTRA_EXPORTED_RUNTIME_METHODS=["'cwrap', 'getValue'"]
-fno-rtti
-fno-exceptions
-o QuickSort.html

```

接下来使用浏览器来运行该应用，在开发者工具面板的“Network”选项卡中可以看到，应用当前加载的模块二进制文件的大小为 150KB，与之前相比有所减小。当多次运行应用后，我们截取了处于中间水平的性能测试结果，如图 7-11 所示。可以看到，相比于最初没有经过任何代码优化的 Wasm 应用，现在无论是在 LT 资源加载阶段还是在 ET 代码执行阶段所消耗的时间，均有大幅度的下降。由此可见代码优化的必要性。

11:07:00.323 Found module in wasm cache: QuickSort	VM6519 QuickSort.js:10
11:07:00.345 11 24 36 42 4 25 9 1 0	QuickSort.html:1237
11:07:00.346 0 24 36 42 4 25 9 1 11	QuickSort.html:1237
11:07:00.347 0 4 9 1	QuickSort.html:1237
11:07:00.348 0 1 9 4	QuickSort.html:1237
11:07:00.349 0 1 4 9 11 25 36 42 24	QuickSort.html:1237
11:07:00.350 0 1 4 9 11 24 36 42 25	QuickSort.html:1237
11:07:00.351 0 1 4 9 11 24 25 42 36	QuickSort.html:1237
11:07:00.352 ▶ (9) [0, 1, 4, 9, 11, 24, 25, 36, 42]	VM6519 QuickSort.js:49
11:07:00.356 Application ET: 115 ms	VM6519 QuickSort.js:49
11:07:00.357 Application LT: 438 ms	VM6519 QuickSort.js:49
11:07:00.358 Application TT: 480 ms	VM6519 QuickSort.js:49

图7-11 上述Wasm应用在经过多道优化工序后的性能测试结果

7.2 使用标准库与文件系统

我们知道，在构建传统 C/C++应用时，可以使用标准库中内置的一系列接口来完成与文件 I/O 相关的操作。比如创建一个新的文件、从文件中读取数据，以及向文件末尾追加数据等。那么，当通过 emcc 来编译一段包含有底层文件系统相关 API 的 C/C++源代码时，Emscripten 在其内部会怎样来模拟这些只有在物理机中才能够使用的应用程序接口呢？

通常来说，基于 C/C++构建的原生应用与基于 JavaScript 构建的 Web 应用分别使用完全不同的文件访问规范。其中基于 C/C++构建的原生应用会直接使用 libc 和 libc++标准库中的同步文件 API 来访问宿主主机中的物理文件系统；而基于 JavaScript 构建的 Web 应用则由于其特殊的运行环境限制，只能通过 JavaScript 在浏览器中异步地访问远程文件资源。并且由于其运行

在浏览器所提供的沙箱环境中，因此也无法直接访问宿主机的物理文件系统。为了解决这些问题，Emscripten 构建了一套与物理文件系统几乎完全相同的虚拟文件系统。这使得我们可以在几乎不需要修改应用源代码的情况下，直接将使用了同步文件 API 的原生应用无痛地移植到基于 Emscripten 工具链构建的特殊 Web 运行时（Runtime）环境中来运行。

在详细介绍这套虚拟文件系统之前，我们先来了解一下 Emscripten 究竟是怎样将使用了标准库的 C/C++ 代码编译成一个基于 WebAssembly/ASM.js 的 Web 端应用的，而 Wasm 模块内部各类 OpCode 所对应的虚拟指令与标准库中相应的代码实现又有着怎样的对应关系。

7.2.1 使用基于 musl 和 libc++ 的标准库

Emscripten 在其内部分别选择使用 musl 和 libcxx 来作为对应的 C 和 C++ 标准库实现。其中 musl 是一种经常被使用在基于 Linux 内核构建的系统上的 C 标准库；而 libcxx 则是一种专门针对 C++ 11 标准实现的 C++ 标准库，该标准库在经过一段时间的发展后，现在也开始支持完整的 C++ 14 特性与部分 C++ 17 特性。接下来，我们将以一个简单的应用示例作为切入点，来了解 emcc 在编译标准库代码时进行的编译策略选择过程。该应用对应的 C/C++ 代码如下：

```
c_standard.cc
#include <emscripten.h>
// C 数学标准库
#include <cmath>
// C++ 输入/输出流标准库
#include <iostream>

using namespace std;

extern "C" {
    int EMSCRIPTEN_KEEPALIVE m_sqrt (int x) {
        return sqrt(x);
    }

    double EMSCRIPTEN_KEEPALIVE m_asin (double x) {
        return asin(x);
    }

    void EMSCRIPTEN_KEEPALIVE m_print () {
        cout << "Hello, WebAssembly!" << endl;
    }
}
```

在这段代码中，一共定义了三个将从 Wasm 模块中导出的方法，它们在其内部又分别调用了来自 C/C++ 标准库中与数学运算相关的 `sqrt` 和 `asin` 函数，以及与终端输入/输出相关的 `cout` 对象。接下来，我们将编写上层 JavaScript 脚本代码，来在浏览器环境中调用这些方法。

```
post-script.js
__ATPOSTRUN__.push(function() {
  console.log(Module['_m_sqrt'](10)); // 3
  console.log(Module['_m_asin'](0.5)); // 0.5235987755982989
  Module['_m_print'](); // Hello, WebAssembly!
});
```

在编译该 Wasm 应用时，需要为编译命令添加“-Os”参数，以便让编译器将 Wasm 模块以及“胶水”脚本文件中多余的调试代码移除。完整的编译命令如下：

```
emcc c_standard.cc
-o c_standard.html
-s WASM=1
-Os
--post-js post-script.js
```

当编译完成后，我们可以在浏览器中运行该应用。实际上，Emscripten 在编译这三个标准的库函数时，分别采用了三种不同的编译策略来将它们移植到 Web 平台上。下面我们将分别介绍这三种不同的编译策略。

编译到 Wasm 虚拟指令

所谓的“编译到 Wasm 虚拟指令”，是指将能够对应到特定 Wasm 虚拟指令的 C/C++ 代码直接进行转译。比如对于上面示例中定义的 `m_asin` 方法，该方法在其内部使用了 C 数学标准库中的 `asin` 函数。如图 7-12 所示，该函数的内部实现是由一系列基本的数学运算代码组成的，而这些代码可以直接被 Emscripten 转译成符合 Wasm 语法规范的二进制 OpCode 代码。比如在 C/C++ 代码中，“1+1”这个数学表达式可以被直接转译成 WAT 下的“(i32.add (i32.const 1) (i32.const 1))”语句。而其所对应的 OpCode 代码为“0x41 0x1 0x41 0x1 0x92”。

直接使用 Wasm 标准中的虚拟指令进行替换

这种编译策略并不是单纯对函数的实现代码进行相应的 WAT 转译，而是会用单个虚拟指令来对整个函数进行替换。比如对于在上面示例中使用的定义在 C 数学标准库中的 `sqrt` 函数，如图 7-13 所示，该函数在标准库中的具体实现也同样包含了一系列复杂的数学运算。但 Emscripten 在编译使用了该函数的 C/C++ 代码时，却完全没有按照标准库中该函数的实现细节来进行相应的代码转译。这是由于 Wasm 在其标准中制定了 `f32.sqrt` 虚拟指令，该指令将会被作为最基本的

“原子操作”直接替换整个源代码中对应标准库函数的调用过程。

```
emscripten/system/lib/libc/musl/src/math/asin.c
Lines 67 to 78 in bf0bdc5

67     double asin(double x)
68     {
69         double z,r,s;
70         uint32_t hx,ix;
71
72         GET_HIGH_WORD(hx, x);
73         ix = hx & 0x7fffffff;
74         /* |x| >= 1 or nan */
75         if (ix >= 0x3ff00000) {
76             uint32_t lx;
77             GET_LOW_WORD(lx, x);
78             if ((ix-0x3ff00000 | lx) == 0)
```

图7-12 musl标准库中asin函数的代码实现细节

```
emscripten/system/lib/libc/musl/src/math/sqrt.c
Lines 83 to 93 in bf0bdc5

83     double sqrt(double x)
84     {
85         double z;
86         int32_t sign = (int)0x80000000;
87         int32_t ix0,s0,q,m,t,i;
88         uint32_t r,t1,s1,ix1,q1;
89
90         EXTRACT_WORDS(ix0, ix1, x);
91
92         /* take care of Inf and NaN */
93         if ((ix0&0x7ff00000) == 0x7ff00000) {
```

图7-13 musl标准库中sqrt函数的代码实现细节

如图 7-14 所示，这里定位到之前示例中 `m_sqrt` 函数在 Wasm 模块内部的代码（WAT）细节。可以看到，Emscripten 直接使用 `f32.sqrt` 虚拟指令来完成整个函数的基本功能，而并没有按照 C 标准库中对应 `sqrt` 函数的复杂实现来进行代码转译。

```
(func $244 (type $3) (param $var$0 i32) (result i32)
  (i32.trunc_s/f64
    (f64.sqrt
      (f64.convert_s/i32
        get_local $var$0
      )
    )
  )
)
```

图7-14 `m_sqrt`函数在Wasm模块内部的代码（WAT）细节

主要功能由 Emscripten 运行时环境提供

在前两种编译策略中，Emscripten 会直接使用在 WebAssembly 标准中定义的虚拟指令集来完成 C/C++代码到 Web 平台的功能“移植”过程。但是对于某些需要依赖特定运行时环境才能够执行的代码来说，单单通过语言层面所提供的语法特性是无法满足其执行要求的。比如在上述示例中使用的定义在 C++标准库中的 `cout` 对象，通过该对象我们可以向当前系统的标准输出流传入数据，而这些数据通常会在应用程序运行时直接反馈到系统的命令行终端中。

为此，Emscripten 在其内部提供了一个特定的基于 JavaScript 语言和上层 Web 浏览器构建的 Emscripten 运行时环境，通过该环境我们便可以在浏览器中模拟出与宿主环境保持一致的功能和底层接口。如图 7-15 所示为从模块中导出的“`_m_print`”函数在上层 Web 浏览器中被调用时其所经过的各函数调用节点（栈）。可以看到，该函数虽然是直接从 Wasm 模块中导出的，但是其在实际执行时，却是通过 Emscripten 运行时环境提供的系统调用（`__syscall`）函数，间接地调用了定义在该运行时环境中的上层接口（`Module.print`）。因此，对于这类依赖于特定宿主环境（比如访问文件系统、播放声音、屏幕绘图）执行的 C/C++代码，它们在被转译到 Web 平台上时，需要使用由 Emscripten 运行时环境提供的相应底层接口来模拟代码所描述的具体功能。而接下来要介绍的虚拟文件系统便是 Emscripten 运行时环境的一个重要组成部分，它在 Web 平台上完美地模拟出了传统文件系统所具有的各个特征。

Module.print	c_standard.html:1237
put_char	c_standard.js:1
write	c_standard.js:1
write	c_standard.js:1
doWritev	c_standard.js:1
__syscall146	c_standard.js:1
<WASM UNNAMED>	wasm-0009a206-225:62
<WASM UNNAMED>	wasm-0009a206-447:46
<WASM UNNAMED>	wasm-0009a206-169:87
<WASM UNNAMED>	wasm-0009a206-69:19
<WASM UNNAMED>	wasm-0009a206-522:41
<WASM UNNAMED>	wasm-0009a206-532:51
<WASM UNNAMED>	wasm-0009a206-259:42
Module._m_print	c_standard.js:1

图7-15 模块导出函数“_m_print”在上层JavaScript环境中的实际函数调用栈

7.2.2 虚拟文件系统结构

如图 7-16 所示为 Emscripten 虚拟文件系统（VFS）的基本组成结构。通常来说，原生 C/C++

应用会通过调用定义在标准库中的同步 API，来与存放在计算机物理存储设备中的文件资源进行交互。而 Emscripten 在处理这些 API 调用时，会默认使用名为“MEMFS”的虚拟文件系统策略来管理相应的文件资源处理过程。

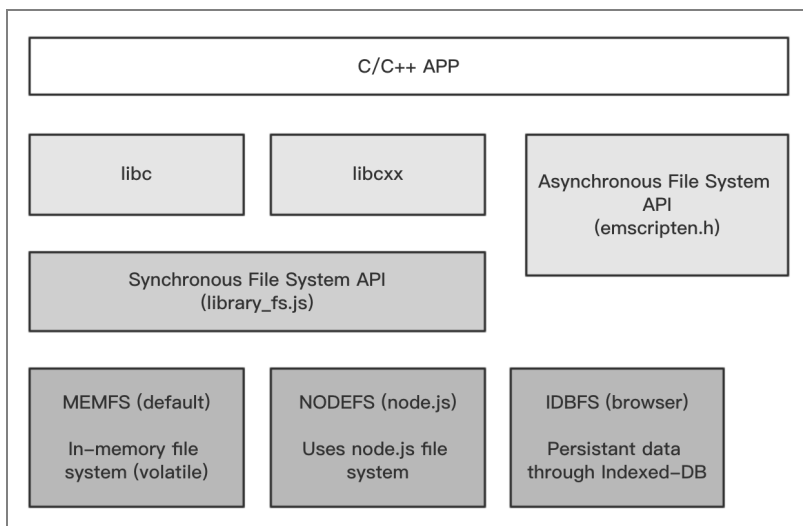


图7-16 Emscripten虚拟文件系统的基本组成结构

基于 MEMFS 策略进行的文件资源处理过程一共分为两步：第一步，在编译应用时，需要为 `emcc` 指定相应的参数来设置虚拟文件系统初始化时包含的文件内容；第二步，当应用编译完成、开始运行并完成资源初始化后，我们便可以通过 Emscripten 运行时环境提供的上层虚拟文件系统接口来操作这些存放在虚拟文件系统中的文件资源。不同于传统的物理文件系统，使用 MEMFS 策略，Emscripten 会将所有通过 `emcc` 指定的初始化文件资源，在应用运行时通过 XHR 远程地加载到浏览器内存中。因此，这些数据会在页面刷新时被重新载入，所有之前写入虚拟文件系统中的数据都将丢失。需要注意的是，应用的主业务流程只有在虚拟文件系统完成内容初始化后才（`__ATPOSTRUN__`）会开始执行。因此，我们说虚拟文件系统的初始化过程是同步进行的，而体积较大的初始化文件将会耗费较长的应用初始化时间。

除了 MEMFS 策略，Emscripten 在其内部还构建了另外两种虚拟文件系统策略。其中名为“NODEFS”的虚拟文件系统策略主要应用在运行于 Node.js 中的 ASM.js 应用，该策略会直接使用 Node.js 的物理文件系统 API 来管理和操作文件资源；名为“IDBFS”的虚拟文件系统策略将基于 IndexedDB 提供一种可以进行持久化的文件数据存储和管理方案。在本书中，我们将主要介绍 MEMFS 和 IDBFS 这两种虚拟文件系统策略的基本使用方法。

7.2.3 打包初始化文件

Emscripten 为我们提供了两种常用的文件打包方式，即“预加载”和“嵌入”。两者的区别在于：预加载方式会将虚拟文件系统初始化时的文件数据放到独立的“.data”文件中进行存储；而嵌入方式则会将初始化文件的内容直接整合到 Emscripten 自动生成的 JavaScript“胶水”脚本文件中。从总体上看，采用独立数据存储文件的预加载方式其虚拟文件的读写效率要远高于嵌入方式。因此，通常我们只会使用嵌入方式来初始化体积较小的文件数据。

emcc 在编译应用时可以通过调用文件打包器来自动打包文件资源，并生成相应的虚拟文件系统调用。通过向编译命令中传递“--preload-file”或“--embed-file”参数，可以让 Emscripten 选择性地使用两种不同的文件打包策略。下面我们通过一个简单的应用示例来了解 Emscripten 虚拟文件系统的基本使用流程。首先给出的是该应用的 C/C++ 代码。

```
file_basic.cc
#include <stdio.h>

int main(int argc, char **argv) {
    // 以只读方式打开一个二进制文件
    FILE *file = fopen("data/core_data.txt", "rb");
    if (!file) {
        printf("Cannot open this file!\n");
        return 1;
    }
    // 读取文件内容
    while (!feof(file)) {
        // 从文件中读取一个字符
        char c = fgetc(file);
        if (c != EOF) {
            // 向终端输出该字符
            putchar(c);
        }
    }
    fclose (file);
    return 0;
}
```

在这段代码中，我们主要做了一件事情，就是从名为“core_data.txt”的文本文件中以字符（字节）为单位来读取数据，然后再将这些字符数据输出到系统终端中。可以看到，在这段代码

中没有使用任何与 Emscripten 有关的方法或宏参数，接下来我们便可以直接通过 `emcc` 命令来将这段原生的 C/C++ 代码无痛地移植到 Web 平台上运行。编译命令语句如下：

```
emcc file_basic.cc
-o file_basic.html
--preload-file data/core_data.txt
-s WASM=1
```

这里首先使用 “`--preload-file`” 参数让 Emscripten 以预加载的方式初始化虚拟文件系统。当命令执行完毕后，Emscripten 会在当前文件夹内生成一个以 “.data” 为后缀的文件，在该文件内则包含所有通过 “`--preload-file`” 参数指定的初始化文件内容（无论是文本文件还是二进制文件，其内容均以二进制编码的形式被存放在该文件内）。Emscripten 生成的 JavaScript “胶水” 脚本文件会在执行时通过 `loadPackage` 方法，并根据编译器记录在代码中的 JSON 元信息（记录了各文件的名称以及其内容在 “.data” 数据文件中的字节偏移位置等信息）来初始化虚拟文件系统。如下所示为我们指定的需要在虚拟文件系统初始化时包含的文件内容。该文件所在的实际路径一定要与 “`--preload-file`” 参数指定的文件全路径名（相对路径+文件名）相符合。

```
data/core_data.txt
Hello, WebAssembly!
```

此时，如果将命令中的 “`--preload-file`” 参数更换为 “`--embed-file`” 参数，当应用编译完成后，则发现 Emscripten 会在 JavaScript “胶水” 脚本文件中直接使用名为 “`fileData[X]`”（其中占位符 “X” 为从数字 “0” 开始依次递增的序列号）的多个数组，以二进制数据的形式来分别保存每一个初始化文件的对应内容。因此，这种虚拟文件系统的初始化文件打包方式会使得 “胶水” 脚本文件的整体体积变大，进而导致该脚本文件所对应的 HTTP 请求消耗的时间变长，模块的初始化过程延后。

另外，我们在通过 “`--preload-file`” 或 “`--embed-file`” 参数指定初始化文件的路径时，可以直接设置一个相对于当前位置的文件夹，这样该文件夹内的所有文件都会被 Emscripten 当作初始化文件，而直接加载到虚拟文件系统中。除此之外，还可以通过在相对路径中加入 “@” 符号来更改文件在虚拟文件系统中的存放位置。比如 “`data/core_data.txt@temp/core_data.txt`” 路径表示将物理文件系统中位于相对位置 “data/” 处的文件 `core_data.txt` 映射到虚拟文件系统中位于 “temp/” 处的同名文件上。当然，在这里映射文件的名称也可以被随意更改。

7.2.4 基本文件系统操作

在之前的应用示例中，我们并没有编写任何用于上层 JavaScript 环境的脚本代码，因此

Emscripten 会自动帮助我们完成虚拟文件系统的初始化过程，并使用 MEMFS 作为整个文件系统根路径 “/” 下的默认文件存储策略。在本节的内容中，我们将介绍如何在上层 JavaScript 代码中手动更换文件系统策略、挂载新的设备，并使用 Emscripten 所提供的虚拟文件系统接口来操作文件资源。

整个 Emscripten 虚拟文件系统的可用 API 全部都是基于 Linux 下 POSIX 文件系统的基本结构而设计的。除某些由于浏览器与本地物理系统环境差异所导致的，无法通过上层 JavaScript 代码来模拟的文件系统特性之外，其余大部分 API 的表现行为均与 POSIX 文件系统中原生接口保持一致，例如 “mkdir” 等 Linux 下与文件系统相关的系统调用接口。

创建并使用虚拟设备

这里可以通过虚拟文件系统提供的特定 API 接口来创建和注册用于处理特殊资源的虚拟设备。这些虚拟设备的表现形式与普通的文件资源一样，通过一个特定的文件路径（挂载点）来表示，而之后所有发生在该路径下的文件读写等操作均会由注册在虚拟设备内部的钩子方法来进行处理。在下面的示例中，我们将着手实现一个“加倍器”设备，该设备会将所有向“挂载点”路径写入的数据所对应的字节数组内的所有数字值均乘以“2”并返回。该示例的完整源代码如下，首先给出的是 C/C++ 部分的代码。

```
file_device.cc
```

```
int main(int argc, char **argv) {  
    return 0;  
}
```

在该示例中，我们不会直接在 C/C++ 代码中来操作文件资源，因此这里只在源代码中放置一个空的 main 函数。而对于虚拟设备的创建、注册及使用等相关流程，则会在上层 JavaScript 代码中，通过使用虚拟文件系统提供的相关 API 来实现。因此，从侧面上看，整个虚拟设备也必须在虚拟文件系统本身完成初始化后才能使用。接下来给出上层 JavaScript 代码。

```
post-script.js
```

```
__ATPOSTRUN__.push(function() {  
    // 创建一个设备 ID ((0) << 8 | (1))  
    var fDevice = FS.makedev(0, 1);  
  
    // 注册设备的回调处理函数  
    FS.registerDevice(fDevice, {  
        open: function (stream) {  
            console.log("[Open] 'DoublingDevice' device stream.");  
        },  
    },
```

```

close: function (stream) {
    console.log("[Close] 'DoublingDevice' device stream.");
},
read: function (stream, buffer, offset, length, position) {
    console.log("[Read] from 'DoublingDevice' device stream.");
},
write: function (stream, buffer, offset, length, position) {
    // 将写入设备的 TypedArray 结构内部的数值加倍并返回
    return buffer.map(function(item) {
        return 2 * item;
    })
}
});

// 在虚拟文件系统内创建一个引用了该设备驱动的设备节点
FS.mkdev('/DoublingDevice', 0777, fDevice);

// 打开一个到该设备实例的流（将调用 open 钩子函数）
var fStream = FS.open('/DoublingDevice', 'w+');

// 向该流写入数据（将调用 write 钩子函数）
var data = new Uint32Array(4);
var _t = FS.write(fStream, new Int32Array([1, 2, 3, 4]), 0, 4, 0);
console.log(_t); // [2, 4, 6, 8]

// 关闭设备流（将调用 close 钩子函数）
FS.close(fStream);
});

```

在这段代码中，从上至下来看，我们主要完成如下 4 件事情。

创建设备 ID

每一个虚拟设备都必须有一个唯一的“设备 ID”，该识别码会在虚拟文件系统内部保持对某个特定设备的唯一引用。因此，首先需要通过 `FS.makedev` 函数来为将要创建的设备实例分配一个全局的、唯一的 ID 标识符。该函数一共接收两个参数，分别表示该设备的“主”与“次”两个部分对应的 ID。该函数在其内部会通过“(主) << 8 | (次)”这个位运算表达式（该表达式表示“主”码每增加“1”就相当于“次”码增加“256”，主/次码整体按照“256”来进位）来计算最终生成的完整设备 ID。

注册设备回调函数

接下来，我们便可以通过 `FS.registerDevice` 函数来注册该设备对文件打开、读写以及关闭等操作的实际处理过程。该函数一共接收两个参数，其中第一个参数为在上一步中生成的虚拟设备 ID；第二个参数为以“对象”形式封装的各类回调处理函数的具体实现。可以看到，这里为该设备实现了 4 种常用的文件资源处理函数。其中，在 `write` 函数中完成了该设备的主要功能逻辑。该函数一共接收如下 5 个参数。

```
/**
 * @param {object} stream - 指向设备实例的流对象
 * @param {ArrayBufferView} buffer - 待写入的数据（类型数组）
 * @param {int} offset - 待写入数据的起始位置
 * @param {int} length - 待写入数据的长度
 * @param {int} position - 写入流对象的偏移量
 */
FS.write(stream, buffer, offset, length[, position])
```

这里我们直接将 `buffer` 类型数组内部的所有元素值均乘以“2”，然后再将计算结果返回给 `FS.write` 函数的调用者。

创建引用节点

现在，我们需要通过 `FS.mkdev` 函数来为该虚拟设备设置一个可供外部调用者进行访问的节点（挂载点）。该函数一共接收三个参数，其中第一个参数为该节点的完整文件路径名称，即访问节点；第二个参数为该节点所具有的权限，与 Linux 系统下的权限表示方式一样，这里我们直接使用“777”这个八进制形式的权限值；第三个参数为将要挂载的虚拟设备 ID。

使用设备

当虚拟设备被挂载到某个可用节点后，我们便可以通过基本的文件操作 API 来使用该设备。同样的，首先需要通过 `FS.open` 函数来获得一个对某文件路径节点的流（文件流/设备流）引用。然后便可以通过 `FS.write` 函数向该流对象中写入数据，虚拟设备在对应的钩子函数内会接收到所传入的数据，经过一系列的数据处理后再将结果返回给调用者。

当上层 JavaScript 代码编写完成后，我们便可以通过如下命令来编译该 Wasm 应用。这里需要注意的是，由于我们没有在 C/C++ 代码中使用任何标准库中与文件系统相关的 API，因此在默认情况下，Emscripten 并不会在“胶水”脚本文件中整合与虚拟文件系统相关的实现代码。为了解决这个问题，我们需要在编译命令中通过“-s FORCE_FILESYSTEM=1”参数来强制要

求 Emscripten 在编译应用时整合虚拟文件系统实现相关的代码到生成的“胶水”脚本文件中。

```
emcc file_device.cc
-o file_device.html
-s WASM=1
--post-js post-script.js
-s FORCE_FILESYSTEM=1
```

当应用编译完成后，我们便可以在浏览器中运行该应用，并观察其运行结果。

设置标准 I/O 流

上面我们介绍了如何在 Emscripten 构建的虚拟文件系统内创建并使用虚拟设备对象。无独有偶，Emscripten 内部的标准输入/输出以及错误输出流便是分别基于“/dev/stdin”、“/dev/stdout”和“/dev/stderr”这三个虚拟设备构建的。我们可以通过调用名为“FS.init”的虚拟文件系统函数来设置上述三者在处理 I/O 流时的具体表现逻辑。一个简单的应用示例如下：

```
file_io.cc
#include <iostream>

using namespace std;
int main(int argc, char **argv) {
    char _t;
    // 调用标准输入流
    cin >> _t;
    // 调用标准输出流
    cout << _t << endl;
    // 调用标准错误输出流
    cerr << _t << endl;
    return 0;
}
```

在上面的 C/C++ 代码中，我们通过标准库中三种不同的输入/输出流对象，分别使用了三种基本类型的 I/O 流。接下来，我们便可以通过 FS.init 函数来设置 Emscripten 在上层 JavaScript 环境中处理这三种 I/O 流时的具体表现形式。

```
post-script.js
__ATPRERUN__.push(function() {
    // 设置与标准 I/O 流相关的回调函数
    FS.init(function() {
        return 64; // "@"
    }, function(ascii_text) {
```

```

    console.log('stdout: ', ascii_text);
  }, function(ascii_text) {
    console.log('stderr: ', ascii_text);
  })
});

```

这里 FS.init 函数一共接收三个匿名函数作为其参数，按照顺序它们依次分别对应于“stdin”（标准输入流）、“stdout”（标准输出流）和“stderr”（标准错误输出流）三者在上层 JavaScript 环境中具体表现形式的实现。其中两个输出流所对应的匿名函数在被调用时，会接收到以 ASCII 码值形式表示的字符序列；而输入流所对应的匿名函数在被调用时，则需要我们向该函数的外部调用者返回以 ASCII 码值形式表示的字符序列。该应用的最终运行结果如图 7-17 所示。

21:36:18.467 stdout 64	file io.js:6476
21:36:18.471 stdout 10	file io.js:6476
21:36:18.472 stderr 64	file io.js:6478
21:36:18.474 stderr 10	file io.js:6478

图7-17 上述Wasm应用在浏览器中的运行结果

在图 7-17 中，数字“64”为字符“@”对应的 ASCII 码值，即通过标准输入流“stdin”所对应匿名函数传递给底层 C/C++代码的字符序列；数字“10”则为“换行符 (LF)”对应的 ASCII 码值，请读者思考一下该字符是在代码中的何处被送入标准输出流中的。

挂载新的文件系统策略

通过为存放在虚拟文件系统上的文件数据“挂载”不同的文件系统策略，其可以使用不同的底层文件系统实现方式来管理文件资源。在默认情况下，Emscripten 会使用 MEMFS 策略来对整个文件系统根目录“/”下的所有文件（夹）和各子文件进行文件管理。在此基础上，我们可以根据不同的业务需求，分别对位于不同路径下的子集文件选用不同的文件系统策略。接下来，我们将以挂载新的 IDBFS 策略为例来编写一个简单的 Wasm 应用。

```

file_mount.cc
int main(int argc, char **argv) {
    return 0;
}

```

类似的，在该应用中同样不需要直接在 C/C++代码中操作文件资源，因此，这里仅在 C/C++文件中留下一个空的 main 函数。

```

post-script.js
__ATPOSTRUN__.push(function() {
    const fileName = '/persistant_data/file.txt';

```

```
// 新建目标文件夹
FS.mkdir('/persistant_data');
// 挂载 IDBFS 文件系统策略
FS.mount(IDBFS, {}, '/persistant_data');

// [IDB -> VFS] 从 IDB 向 VFS 中同步数据
FS.syncfs(true, function (err) {
  // 以追加的方式打开一个文件流
  var stream = FS.open(fileName, 'a+');
  var data = new Uint8Array([65, 66, 67, 68]);
  // 调整读写指针到文件内容的末尾
  FS.llseek(stream, 0, 2);
  // 向文件中追加写入数据
  FS.write(stream, data, 0, data.length);
  FS.close(stream);

  // [VFS -> IDB] 从 VFS 向 IDB 中持久化数据
  FS.syncfs(false, function (err) {
    // 以二进制流的形式读取文件数据
    var buffer = FS.readFile(fileName, {
      encoding: 'binary'
    });
    // 将二进制数据转换为字符串，并打印
    console.log(Module['intArrayToString'](buffer));
  });
});
```

在这段代码中，我们首先使用 `FS.mkdir` 方法创建了一个专门用于挂载 IDBFS 策略的文件夹。所有处在（直接或以子文件/文件夹的形式）该文件夹下的文件资源都会按照 IDBFS 策略规定的实现方式来进行资源存储。接下来，我们便直接通过 `FS.mount` 方法向该文件夹下挂载了相应的文件系统策略（关于该方法中各参数的具体含义，读者可以自行查阅官方文档来了解，这里不做过多介绍）。我们之前介绍过，在 IDBFS 策略下运行的 VFS 可以通过浏览器的 IndexedDB 对其内部的文件资源数据进行持久化处理。同样的，位于 IDB 中的文件数据也可以被恢复到相应的 VFS 内存中，即 VFS 与 IDB 之间可以双向传递数据，如图 7-18 所示。而这便引出了一个问题：我们应该何时让 IDB 中的数据同步到 VFS 内存中进行资源初始化，而又应该何时让 VFS 内存中的数据被同步到 IDB 中进行持久化呢？

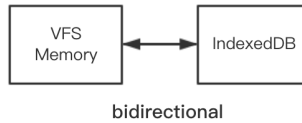


图7-18 在VFS与IDB之间双向传递数据

为了解决这个问题，Emscripten 运行时环境为我们提供了名为“FS.syncfs”的方法，通过该方法，我们可以直接在上层 JavaScript 代码中决定文件资源在 VFS 与 IDB 之间的传递方向。该方法一共接收两个参数，如下所示。

```
/**
 * @param {boolean} populate - 该参数若为 true，则表示从 IDB 向 VFS 中同步数据；否则，将从 VFS 向
IDB 中持久化数据
 * @param {function} callback - 初始化完毕执行的回调函数
 */
FS.syncfs(populate, callback);
```

可以看到，实际上通过设置该方法的第一个参数，便可以决定文件资源数据在 VFS 与 IDB 之间的传递方向。当数据在某一个方向上完全传递完毕（存放到内存或 IDB 中）之后，该方法第二个参数对应的匿名函数将会被调用。

在上面的示例中，我们首先尝试从 IDB 向 VFS 内存中恢复已有的数据。当数据恢复完成后，我们直接对当前 VFS 内存中的文件数据进行修改——在该文件（/persistant_data/file.txt）资源的尾部追加了部分内容，然后再次通过 FS.syncfs 方法将该文件的内容持久化到 IDB 中。如此一来所达到的效果是，只要每一次刷新当前应用所在的 Web 页面时，持久化在 IDB 中的对应文件其内容便会不断增多。最后，我们可以通过如下命令来编译该应用。

```
emcc file_mount.cc
-o file_mount.html
-s WASM=1
--post-js post-script.js
-s FORCE_FILESYSTEM=1
-s EXTRA_EXPORTED_RUNTIME_METHODS=["'intArrayToString'"]
```

创建、读写和检索文件

这里将介绍除上述 VFS 的主要 API 接口外，可用在上层 JavaScript 环境中的其他常用文件操作接口。下面我们直接来看一个综合的应用示例。

```
file_others.cc

int main(int argc, char **argv) {
```

```
    return 0;
}
```

同样的，在该应用中，我们并不会直接在 C/C++ 代码中操作文件资源。

```
post-script.js
```

```
__ATPOSTRUN__.push(function() {
  // 新建文件夹
  FS.mkdir('/data');

  // 向文件中写入内容
  FS.writeFile('/data/file', 'Hello, WebAssembly!');

  // 从文件中读取内容
  console.log(FS.readFile('/data/file', { encoding: 'utf8' })); // “Hello, WebAssembly!”

  // 显示文件信息
  console.log(FS.stat('/data/file'));

  // 重命名文件
  FS.rename('/data/file', '/data/fileNew');

  // 分割文件内容
  FS.truncate('/data/fileNew', 5);
  console.log(FS.readFile('/data/fileNew', { encoding: 'utf8' })); // “Hello”

  // 打开一个文件流
  var stream = FS.open('/data/fileNew', 'w+');

  // 判断该流节点类型
  console.log(FS.isFile(stream.node.mode)); // true

  // 向文件中写入内容（二进制形式）
  var buffer = new Uint8Array('Hello, YHSPY!'.split('').map(item => item.charCodeAt()));
  FS.write(stream, buffer, 0, buffer.length, 0);

  // 从文件中读取内容（二进制形式）
  var buffer = new Uint8Array(13);
  FS.read(stream, buffer, 0, buffer.length, 0);
  console.log(
    Array.from(buffer).map(item => String.fromCharCode(item)).join('')
```



```

); // “Hello, YHSPY!”
// 获得当前的工作目录
console.log(FS.cwd()); // “/”

// 创建一个软链接
FS.symlink('/data/fileNew', 'link');

// 读取该软链接的真实路径
console.log(FS.readlink('link')); // “/data/fileNew”

// 关闭文件流
FS.close(stream);
});

```

在上面的代码中，我们通过 Emscripten 构建的 VFS 其内部封装的一系列上层 JavaScript 接口完成了与文件系统相关的多种操作。关于其中的大部分 API，都可以在 Emscripten 的官方文档中找到对应的使用方法，这里不再展开介绍。而值得关注的一个地方是：与 Linux 系统类似，Emscripten 也在 VFS 内部使用了类似的“位掩码”来作为区分各文件流类型的标志。比如最常见的普通文件（S_IFREG）流，其位掩码为字面量“0100000”所代表的八进制值。而在上面代码中使用的 FS.isFile 函数，便正是通过各流节点中的掩码值来判断该流所对应的实体类型的。

7.2.5 懒加载

在前面的应用示例中，我们介绍过在编译使用文件系统相关接口的 C/C++代码时，需要通过指定“--embed-file”或“--preload-file”参数，来让 Emscripten 虚拟文件系统在应用初始化时将 C/C++代码中使用到的文件资源提前加载到 VFS 的内存中。而这种方式存在的问题是：一次性完全加载数量众多的文件资源会在一定程度上降低 Wasm 应用的首屏加载效率。为了解决这个问题，Emscripten 运行时环境提供了一个名为“FS.createLazyFile”的方法，通过该方法，我们可以在应用代码实际使用到某个文件资源时，再通过 XHR 从远程服务器中异步获取该文件的内容。从总体上看，这便是一种资源懒加载的过程。

为了能够在上层 JavaScript “胶水”脚本文件中尽可能地模拟在 C/C++代码中操作文件资源时的同步（Synchronous）过程，Emscripten 便直接在 FS.createLazyfile 方法内部使用了同步的 XHR 请求来获取文件资源，而这又带来了另一个问题。

如图 7-19 所示，由于同步的 XHR 请求会阻塞线程的执行，因此 Chrome、Firefox 等常见浏览器均已在其最新的版本中，禁止在用于控制页面绘制与响应的主线程中发送同步的 XHR 请

求。所以，在使用该 API 时，我们便需要将其放置在独立的 Web Worker 线程中来执行，并且还需要在应用的主线程和 Worker 线程之间建立一套专门用于消息与事件传递的沟通机制。

Note: Starting with Gecko 30.0 (Firefox 30.0 / Thunderbird 30.0 / SeaMonkey 2.27), Blink 39.0, and Edge 13, synchronous requests on the main thread have been deprecated due to the negative effects to the user experience.

图7-19 浏览器禁止在主线程中发送同步的XHR请求

基于 Worker 线程实现的资源懒加载过程如图 7-20 所示。我们需要手动编写用于加载 Worker 脚本的 HTML 页面，并借助 Emscripten 工具链来构建包含有完整 VFS 实现的 Worker 脚本。

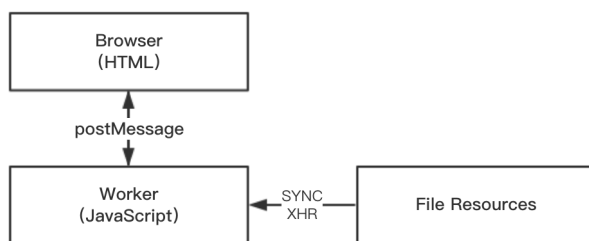


图7-20 基于Worker线程实现的资源懒加载过程

首先给出示例应用的 C/C++ 代码。

```

file_lazy.cc
#include <iostream>
#include <fstream>
#include <emscripten.h>

using namespace std;
// 待读取文件的文件名
const char* fileName = "/core_data.txt";
// 用于在上层 JavaScript 环境中进行异步调用的导出函数
extern "C" void EMSCRIPTEN_KEEPALIVE fetchAndPrintFileContent (void) {
    string line;
    ifstream file(fileName);
    if (file.is_open()) {
        // 读取一行文件内容
        while (getline(file, line)) {
            // 打印读取到的文件内容
            cout << line << endl;
        }
    }
}
  
```

```

    file.close();
} else {
    cerr << "Unable to open file!" << endl;
}
}

```

在上面的代码中，我们定义了一个将会被导出到上层 JavaScript 环境中的函数，该函数在其内部通过标准库提供的文件系统 API 读取了“/core_data.txt”文件的内容，并将其逐行打印出来。接下来，我们将构建用于懒加载文件资源的 Worker 脚本，该脚本的内容一共分为三个部分：第一部分为使用 createLazyFile 方法设置文件懒加载的代码，在这部分代码中，我们需要根据 C/C++ 代码中使用到的所有文件信息，来设置和绑定相应的文件懒加载过程；第二部分为从 Worker 线程向主线程传递数据的通信代码，在这里我们主要重写（使用 postMessage 将待打印内容传递到主线程）了“胶水”脚本文件中的 print 方法，该方法将会作为映射函数来同步 C/C++ 代码向外部打印消息的动作；在第三部分代码中，我们将主动调用从 C/C++ 代码中导出的文件读取函数（Module['fetchAndPrintFileContent']）来激活整个文件的懒加载过程。

首先给出前两部分对应的 JavaScript 代码。其中 FS.createLazyFile 方法在调用时一共接收如下 5 个参数。

```

/**
 * @param {string/object} parent - 目标文件在 VFS 中的父路径名
 * @param {string} name - 目标文件在 VFS 中的名称
 * @param {string} url - 目标文件的远程物理位置
 * @param {boolean} canRead - 文件是否可读
 * @param {boolean} canWrite - 文件是否可写
 */
FS.createLazyFile(parent, name, url, canRead, canWrite);

```

需要注意的是，该方法需要被放置在 Emscripten 运行时环境的 preInit 回调函数（是一种生命周期钩子函数）中才能够被正常执行。在懒加载文件设置完毕后，我们接着又重写了 Module 对象中的 print 方法，这里直接将该方法接收到的消息通过 postMessage 传递给浏览器的主线程，后续我们将在对应的 HTML 文件中接收和处理这部分消息。

```

worker_pre_js.js

if (typeof(Module) === "undefined") Module = {};
// 将 FS.createLazyFile 函数的调用流程挂载到 preInit 阶段
Module["preInit"] = function() {
    FS.createLazyFile(
        '/',

```

```

    "core_data.txt",
    "http://localhost:8888/data/core_data.txt",
    true,
    false
  );
};
// 向浏览器主线程传递消息（该函数接收到的消息由 C++ 代码中的 cout 对象输出）
Module["print"] = function(s) {
  self.postMessage({
    channel: "stdout",
    line: s
  });
};

```

接下来，我们在 “__ATPOSTRUN__” 生命周期队列中加入应用的上层业务代码。这里直接通过调用从 C/C++ 代码中导出的 `fetchAndPrintFileContent` 函数来激活文件的懒加载过程。

post-script.js

```

__ATPOSTRUN__.push(function() {
  // 异步调用从 C/C++ 代码中导出的函数
  setTimeout(function() {
    Module['_fetchAndPrintFileContent']();
  }, 2000);
});

```

最后，我们需要在一个 HTML 文件中来整合应用的资源和主流程。如下面代码所示，我们在该文件中加载并启动了 Worker 线程，同时为该 Worker 线程指定了用于接收消息的回调方法。在该方法中，会直接将所收到的消息打印到浏览器控制台中。

file_lazy.html

```

<!doctype html>
<html>
<head><meta charset="utf-8"><title>File Lazy Loading</title></head>
<html>
<body>
  <script>
    // 加载并运行 Worker 线程
    var worker = new Worker('./file_lazy_worker.js');
    // 接收从 Worker 线程传递过来的消息
    worker.onmessage = function(event) {
      if (event.data.channel === "stdout") {

```

```

        console.log(event.data.line);
    }
};
</script>
</body>
</html>

```

用于编译该 Wasm 应用的命令语句如下：

```

emcc file_lazy.cc
-o file_lazy_worker.js
-s WASM=1
--pre-js worker_pre_js.js
--post-js post-script.js

```

7.2.6 Fetch API

除可以直接在上层 JavaScript 代码中通过 VFS 提供的相关接口来加载位于远程位置的文件资源以外，Emscripten 还提供了一套可以应用在 C/C++ 代码中的“Fetch API”接口。通过这套接口，我们便可以在 C/C++ 代码中以回调函数的形式来异步地加载和处理文件资源。这些原生接口的调用过程在经过 emcc 的编译后，会交由 Emscripten 运行时环境来负责生成实际的由上层 JavaScript 代码实现的文件资源请求逻辑。因此，这些 API 的实际执行过程也将会受到浏览器 CORS 及同源策略的安全限制。一个简单的应用示例如下：

```

file_fetch.cc
#include <iostream>
#include <string.h>
#include <emscripten/fetch.h>

using namespace std;
// 文件资源加载成功后执行的回调函数
void downloadSucceeded(emscripten_fetch_t *fetch) {
    // 打印读取到的文件内容
    for (int i = 0; i < fetch->numBytes; i++) {
        cout << fetch->data[i];
    }
    cout << endl;
    // 释放请求结果所对应的结构体对象
    emscripten_fetch_close(fetch);
}

```

```
// 文件资源加载失败后执行的回调函数
void downloadFailed(emsripten_fetch_t *fetch) {
    cout << "Downloading failed, HTTP failure status code: " << fetch->status << endl;
    emsripten_fetch_close(fetch);
}

int main(int argc, char **argv) {
    // 声明一个“请求属性”结构体
    emsripten_fetch_attr_t attr;
    // 初始化，并为该结构体分配内存空间
    emsripten_fetch_attr_init(&attr);
    // 配置该“请求属性”结构体（设置 HTTP 请求类型、回调函数及文件系统策略）
    strcpy(attr.requestMethod, "GET");
    attr.attributes = EMSRIPTEN_FETCH_LOAD_TO_MEMORY;
    // 绑定回调函数
    attr.onsuccess = downloadSucceeded;
    attr.onerror = downloadFailed;
    // 发送请求，获取文件资源
    emsripten_fetch(&attr, "http://localhost:8888/data/core_data.txt");
}
```

在这段代码中，我们可以很直观地看出各个 API 的基本使用方法。这里首先要介绍的是名为“`emsripten_fetch_attr_t`”的结构体。该结构体用于表示一次 HTTP 请求所包含的基本状态和行为等信息，比如最常见的 HTTP 请求类型、各种请求状态下对应的回调函数，以及该次请求所携带的数据内容等一系列参数。如下是该结构体的完整定义，以及对各参数字段的简单说明。

```
typedef struct emsripten_fetch_attr_t {
    char requestMethod[32]; // HTTP 请求类型（常规的 GET、PUT 以及 EM_IDB_STORE 等类型）
    void *userData; // 自定义数据
    // 各类回调函数
    void (*onsuccess)(struct emsripten_fetch_t *fetch); // 成功
    void (*onerror)(struct emsripten_fetch_t *fetch); // 失败
    void (*onprogress)(struct emsripten_fetch_t *fetch); // 进行中
    uint32_t attributes; // 请求行为属性
    unsigned long timeoutMSecs; // 指定请求超时时间（毫秒）
    EM_BOOL withCredentials; // 是否允许在 CORS 下携带 Cookie 信息
    const char *destinationPath; // 指定 IDB 中的资源持久化路径
    const char *userName; // 指定用于身份验证的用户名
    const char *password; // 指定用于身份验证的密码
    const char * const *requestHeaders; // 指向一个包含 Header 信息的数组结构
```

```
const char *overriddenMimeType; // 设置响应数据的 MIME 类型
const char *requestData; // 请求数据（字符串形式）
size_t requestDataSize; // 请求数据的长度
} emscripten_fetch_attr_t;
```

在上面的定义中，其中一个值得关注的是 **attributes** 参数。该参数包含了一些可以决定请求体和请求结果状态的子参数。这些子参数通过“|”（按位或）的形式进行连接并传递给“请求属性”结构体的 **attributes** 属性，进而一同发挥作用。该参数的可用子参数为如下常量值。

```
// 将请求的响应结构体直接存放在浏览器内存中
#define EMSCRIPTEN_FETCH_LOAD_TO_MEMORY 1

// 将在请求数据传输过程中产生的中间进度数据传递给名为 onprogress 的回调函数
#define EMSCRIPTEN_FETCH_STREAM_DATA 2

// 将请求的响应结构体持久化在浏览器的 IDB 中
#define EMSCRIPTEN_FETCH_PERSIST_FILE 4

// 对资源进行基于 IDB 的断点续存
#define EMSCRIPTEN_FETCH_APPEND 8

// 使用新的请求资源替换 IDB 中当前的持久化资源
#define EMSCRIPTEN_FETCH_REPLACE 16

// 首先尝试从 IDB 中读取数据，若无法找到，再从远程服务器拉取
#define EMSCRIPTEN_FETCH_NO_DOWNLOAD 32

// 以同步的形式来发送和处理 HTTP 请求
#define EMSCRIPTEN_FETCH_SYNCHRONOUS 64

// 使用 Waitable 模式来处理 HTTP 请求
#define EMSCRIPTEN_FETCH_WAITABLE 128
```

读者可以自行搭配使用这些子参数，限于篇幅，这里不再展开介绍。另一个值得关注的是名为“**emscripten_fetch_t**”的结构体。该结构体在其内部存放了所有用于表示请求结果的相关字段。下面给出该结构体的完整定义，以及对各参数的基本说明。

```
typedef struct emscripten_fetch_t {
    unsigned int id; // 该次请求的唯一标识符
    void *userData; // 请求传递过来的自定义数据
    const char *url; // 远程资源的地址
```

```

const char *data; // 包含了响应结果数据
uint64_t numBytes; // 包含了响应结果数据的字节长度/分段长度
uint64_t dataOffset; // 包含了已经下载的字节数量
uint64_t totalBytes; // 响应体数据的总长度
unsigned short readyState; // 当前 XHR 请求的状态
unsigned short status; // 响应的 HTTP 状态码
char statusText[64]; // 可读形式的状态响应码

uint32_t __proxyState;

// For internal use only.
emscripten_fetch_attr_t __attributes;
} emscripten_fetch_t;

```

接下来，我们通过如下命令来编译该应用。这里需要为 `emcc` 提供另外的 “-s FETCH=1” 参数，以便让 Emscripten 运行时环境能够整合 Fetch API 所需要使用的上层 JavaScript 代码实现。

```

emcc file_fetch.cc
-o file_fetch.html
-s WASM=1
-s FETCH=1

```

下面我们将通过几个典型的示例来介绍 Fetch API 的常见使用场景。关于各个 API 更详细的使用方法，读者可以参考官方文档中的相关说明。

跟踪资源下载进度

我们可以通过为 `emscripten_fetch_attr_t` 结构体指定 `onprogress` 回调函数的方式来实时跟踪当前文件资源的下载进度。该示例的 C/C++ 代码如下：

```

file_fetch.cc

#include <iostream>
#include <emscripten/fetch.h>

using namespace std;

void downloadProgress(emscripten_fetch_t *fetch) {
    // 结构体中的 totalBytes 属性依赖服务器在响应头中返回的 Content-Length 字段，因此其值可能为空
    if (fetch->totalBytes) {
        // dataOffset 属性表示当前已下载的字节数量
        cout << "Total(%): " << fetch->dataOffset * 100.0 / fetch->totalBytes << endl;
    } else {
        cout << "Total(Bytes): " << fetch->dataOffset << endl;
    }
}

```



```

    }
}

int main(int argc, char **argv) {
    emscripten_fetch_attr_t attr;
    emscripten_fetch_attr_init(&attr);
    strcpy(attr.requestMethod, "GET");
    attr.attributes = EMSCRIPTEN_FETCH_LOAD_TO_MEMORY;
    // 绑定 onprogress 回调函数
    attr.onprogress = downloadProgress;
    emscripten_fetch(&attr, "http://localhost:8888/data/core_data.txt");

    return 0;
}

```

基于 IDB 缓存远程/本地资源

我们可以通过为 `emscripten_fetch_attr_t` 结构体中的 `attributes` 属性指定名为“`EMSCRIPTEN_FETCH_PERSIST_FILE`”的参数，来让 `Fetch` API 将从远程位置获取到的文件资源直接持久化在本地浏览器的 `IndexedDB` 数据库中。示例代码如下：

```

int main(int argc, char **argv) {
    emscripten_fetch_attr_t attr;
    emscripten_fetch_attr_init(&attr);
    strcpy(attr.requestMethod, "GET");
    // 启用 IDB 缓存策略，并用新获取到的文件数据替换 IDB 中的已有数据
    attr.attributes = EMSCRIPTEN_FETCH_REPLACE | EMSCRIPTEN_FETCH_PERSIST_FILE;
    emscripten_fetch(&attr, "http://localhost:8888/data/core_data.txt");

    return 0;
}

```

上述代码会将将从远程位置获取到的文件资源直接持久化存储到当前浏览器的 `IDB` 中。同样的，我们也可以借助 `Fetch` API 将自定义的本地数据持久化，只不过这里需要将“请求属性”结构体中的 `requestMethod` 属性（表示请求类型）指定成名为“`EM_IDB_STORE`”的本地数据持久化请求。示例代码如下：

```

int main(int argc, char **argv) {
    // 自定义数据
    uint8_t data[] = {'A', 'B', 'C', 'D'};
}

```

```
emscripten_fetch_attr_t attr;
emscripten_fetch_attr_init(&attr);
// 设置请求类型为本地的 IDB 缓存资源请求
strcpy(attr.requestMethod, "EM_IDB_STORE");
// 设置文件存储策略
attr.attributes = EMSCRIPTEN_FETCH_REPLACE | EMSCRIPTEN_FETCH_PERSIST_FILE;
// 设置待存储数据的内容和长度
attr.requestData = (char *)data;
attr.requestDataSize = sizeof(data) / sizeof(char);
emscripten_fetch(&attr, "localStorage.dat");
}
```

7.3 处理浏览器事件

基于 Emscripten 在其 `html5.h` 头文件中封装的接口，我们可以直接在 C/C++ 代码中使用上层浏览器的原生 HTML 事件系统。Emscripten 会在其运行时环境中通过 JavaScript “胶水”脚本将本地 C/C++ 代码与浏览器环境进行绑定。该头文件中提供的大多数 API 均使用了基于事件驱动的设计架构，即通过注册一个将在事件发生时被调用的回调函数，来间接地访问该事件发生时的相关信息。示例代码如下：

```
#include <iostream>
#include <emscripten/html5.h>

using namespace std;
// 设置事件回调函数
EM_BOOL em_keyboard_event_callback (
    int eventType,
    const EmscriptenKeyboardEvent *e,
    void *userData
) {
    cout << e->key << endl;
    return EM_TRUE;
}

int main (int argc, char **argv) {
    // 设置事件注册函数
    emscripten_set_keypress_callback(
        "#window",
        nullptr,
```

```
EM_FALSE,
    &em_keyboard_event_callback
);
return 0;
}
```

在上面的代码中，我们通过 `emscripten_set_keypress_callback` 方法为 HTML 5 中的键盘事件注册了相应的回调函数。该函数会在用户按下键盘上的按键时被调用，并将对应的按键名直接打印出来。如图 7-21 所示为该应用在宏观层面的基本交互逻辑。可以看到，用于构建该应用的 C/C++ 代码主要由三部分组成：用于绑定事件发生过程的事件注册函数、用于接收并处理事件发生消息的事件回调函数，以及用于辅助连接 Emscripten 运行时环境的事件封装类型。

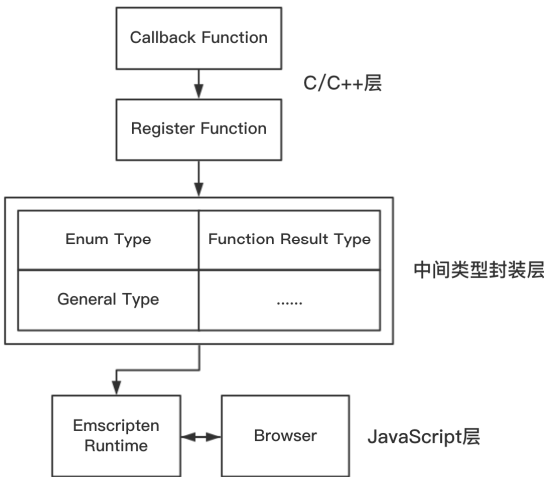


图7-21 上述示例应用在宏观层面的基本交互逻辑

7.3.1 事件注册函数

一个典型的事件注册函数其组成结构如下：

```
EMSCRIPTEN_RESULT emscripten_set_[some]_callback(
    const char *target, // 待“观察”元素的标识
    void *userData, // 用户自定义数据（将会被传递到事件回调函数中）
    EM_BOOL useCapture, // 是否使用事件捕获模式
    em_[someevent]_callback_func callback // 事件发生时的回调函数
);
```

其中，第一个参数“`target`”表示需要注册事件处理函数的 HTML 元素的 id 值。该参数可

以选取如下一系列值。

- `0 / NULL / nullptr`：表示根据事件类型自动选取默认的元素。
- `"#window"`：表示浏览器最上层环境中的 `window` 对象。
- `"#document"`：表示浏览器最上层环境中的 `window.document` 对象。
- `"#screen"`：表示浏览器最上层环境中的 `window.screen` 对象。
- `"#canvas"`：表示 Emscripten 默认使用的 WebGL 画布元素。
- `"[id]"`：表示页面中无前置 `"#"` 符号并标记了特定 `"id"` 的 HTML 元素。

第二个参数 `"userData"` 表示用户自定义的一簇数据，这些数据将会在相应的事件回调函数被调用时，一同作为参数传递到该函数中。

第三个参数 `"useCapture"` 将会在上层 Emscripten 运行时环境中被直接映射到 `EventTarget.addEventListener` 对象的同名 `useCapture` 属性上。该参数将决定回调函数的触发时机。如果其值为真，则回调函数会在 DOM 对象的捕获和目标阶段被触发；否则，会在目标阶段和 DOM 对象的冒泡阶段被触发。这里在设置该参数值时，需要专门使用 Emscripten 在其内部预定义的宏变量值 `"EM_TRUE"` 和 `"EM_FALSE"`，它们分别对应于逻辑上的“真”和“假”。

最后一个参数 `"callback"` 为指向该事件回调函数的指针。

7.3.2 事件回调函数

用于响应事件的回调函数其基本结构如下：

```
EM_BOOL (*em_[someevent]_callback_func) (  
    int eventType,  
    const Emscripten[Some]Event *someEvent,  
    void *userData  
);
```

该回调函数一共接收三个参数。

第一个参数表示当前所发生事件的具体类型，该事件类型将以“Emscripten 宏常量”的形式来表示。比如宏常量 `"EMSCRIPTEN_EVENT_CLICK"` 表示一次用户鼠标的单击事件；宏常量 `"EMSCRIPTEN_EVENT_KEYPRESS"` 则表示计算机键盘按键被按下时所发生的事件。

第二个参数为一个针对某特定类型事件的结构体对象，在该对象中包含了所有与该事件有关的信息。下面显示的是 `EmscriptenKeyboardEvent` 键盘事件对应响应体的完整结构体定义，可以看到，其中的 `key`、`keyCode` 等属性完整地给出了关于触发该事件的相关按键信息。当然，每一种事件类型都有其不同的事件信息结构体定义。关于这部分内容，读者可以参考官方文档中的详细说明。

```
typedef struct EmscriptenKeyboardEvent {
    EM_UTF8 key[EM_HTML5_SHORT_STRING_LEN_BYTES];
    EM_UTF8 code[EM_HTML5_SHORT_STRING_LEN_BYTES];
    unsigned long location;
    EM_BOOL ctrlKey;
    EM_BOOL shiftKey;
    EM_BOOL altKey;
    EM_BOOL metaKey;
    EM_BOOL repeat;
    EM_UTF8 locale[EM_HTML5_SHORT_STRING_LEN_BYTES];
    EM_UTF8 charValue[EM_HTML5_SHORT_STRING_LEN_BYTES];
    unsigned long charCode;
    unsigned long keyCode;
    unsigned long which;
} EmscriptenKeyboardEvent;
```

最后一个参数为通过事件注册函数传递进来的“`userData`”参数，在该参数中存放了用户自定义的数据。

整个回调函数所返回的布尔值“`EM_BOOL`”将决定事件在经过该方法处理后接下来的行为。当返回值为“`true`”时，表示当前的事件已经被处理，浏览器的默认事件行为将会被制止（`preventDefault`）；而当值为“`false`”时，则表示事件将被继续处理，浏览器会自动执行默认的事件处理方式，并通过冒泡机制继续向外层 DOM 节点传递该事件。这里如果在调用事件注册函数时为回调函数参数传递了空指针，则表示取消执行某节点上的事件处理回调函数。

7.3.3 通用类型与返回值类型

对于在前面示例中使用的如 `EM_BOOL` 等宏常量，实际上它们均是 Emscripten 在其内部封装的具有特定语义的专用数据类型，本节我们将介绍这些数据类型。这里我们将它们分为两类：通用类型和返回值类型。其中，通用类型是指可以在任意的回调函数、事件结构体及注册函数内使用的数据类型；而返回值类型则专门指事件注册/调用函数的返回值类型。

通用类型

如图 7-22 所示，这里 Emscripten 一共定义了 4 种通用类型。

```
emscripten/system/include/emscripten/html5.h
Lines 72 to 75 in bf0bdc5

72  #define EM_BOOL int
73  #define EM_TRUE 1
74  #define EM_FALSE 0
75  #define EM_UTF8 char
```

图7-22 4种Emscripten通用类型

其中，“EM_BOOL”类型为 Emscripten 布尔值的统一类型，该类型还可以被进一步细分为由“EM_TRUE”表示的布尔“真”类型，以及由“EM_FALSE”表示的布尔“假”类型。“EM_UTF8”类型则通常以字节数组的形式来描述一个字符串，一般用于表示标签元素的 id 名称。

返回值类型

如图 7-23 所示，Emscripten 一共定义了 10 种注册函数的返回值类型。其中“EMSCRIPTEN_RESULT”宏常量为所有返回值类型的统一类型。

```
emscripten/system/include/emscripten/html5.h
Lines 58 to 70 in bf0bdc5

58  #define EMSCRIPTEN_RESULT int
59
60
61  #define EMSCRIPTEN_RESULT_SUCCESS          0
62  #define EMSCRIPTEN_RESULT_DEFERRED         1
63  #define EMSCRIPTEN_RESULT_NOT_SUPPORTED    -1
64  #define EMSCRIPTEN_RESULT_FAILED_NOT_DEFERRED -2
65  #define EMSCRIPTEN_RESULT_INVALID_TARGET   -3
66  #define EMSCRIPTEN_RESULT_UNKNOWN_TARGET   -4
67  #define EMSCRIPTEN_RESULT_INVALID_PARAM    -5
68  #define EMSCRIPTEN_RESULT_FAILED          -6
69  #define EMSCRIPTEN_RESULT_NO_DATA          -7
70  #define EMSCRIPTEN_RESULT_TIMED_OUT        -8
```

图7-23 10种注册函数的Emscripten返回值类型

事件注册函数的返回值通常用于表示当前的事件注册过程是否被顺利执行。比如当我们在 C/C++ 代码中尝试通过 Emscripten 来绑定或执行一个浏览器所不支持的事件（比如对 Battery 电池进行相关的状态检测）时，其所对应的事件注册/调用函数便会返回“EMSCRIPTEN_RESULT_NOT_SUPPORTED”宏常量所代表的值。我们可以直接通过该宏常量的名称来了解本次事件绑定过程的异常原因，即：当前浏览器平台并不支持所绑定的事件类型。示例代码如下：

```
#include <iostream>
#include <emscripten/html5.h>

using namespace std;
```

```
int main (int argc, char **argv) {
    EmscriptenBatteryEvent *emscriptenBatteryEvent;
    // 调用当前平台用于获取 Battery 状态的方法
    EMSCRIPTEN_RESULT result = emscripten_get_battery_status(emscriptenBatteryEvent);
    // 特性不被支持时报错
    if (result == EMSCRIPTEN_RESULT_NOT_SUPPORTED) {
        cout << "[Battery] Feature not supported!" << endl;
    }
    // 在退出 Runtime 之前输出缓冲区中的内容
    fflush(stdout);
    return 0;
}
```

需要注意的是，在正常情况下，Emscripten 并不会在 C/C++ 代码退出 main 函数时自动结束其上层的 JavaScript 运行时环境，这就意味着 Emscripten 并不会在此刻立即调用全局的 C++ 对象析构函数，同样也不会刷新位于缓冲区中的“stdio”流数据。因此，为了能够正常地输出存放在缓冲区中的数据，这里需要通过主动调用 fflush 函数的方式来将缓冲区清空，或者在编译应用时，为编译命令添加额外的“NO_EXIT_RUNTIME=0”参数，该参数能够让 Emscripten 运行时环境在 main 函数执行结束后立即退出，进而在清理资源时将缓冲区清空。

7.3.4 常用事件

在本节中，我们将介绍 html5.h 头文件中定义的一些常用事件的基本用法，同时结合前面介绍的 Emscripten 事件系统的基本特性，来构建具有简单功能的 Wasm 应用。

鼠标事件 (Mouse)

示例代码如下：

```
html5_mouse.cc
#include <emscripten/html5.h>
#include <emscripten.h>

using namespace std;
// 鼠标事件的回调函数
EM_BOOL em_mouse_event_callback (
    int eventType,
    const EmscriptenMouseEvent *e,
    void *userData
) {
```

```

    auto x = e->targetX;
    auto y = e->targetY;
    // 将鼠标当前的位置数据传递到上层 JavaScript 环境中
    EM_ASM_({
        let x = $0;
        let y = $1;
        // 根据鼠标位置的两个坐标值来更换网页背景色
        document.querySelector("body").style['backgroundColor'] = `#${x}${y}`;
    }, x, y);

    return EM_TRUE;
}

int main (int argc, char **argv) {
    // 注册鼠标事件
    emscripten_set_mousemove_callback(
        "#window",
        nullptr,
        EM_FALSE,
        &em_mouse_event_callback);
    return 0;
}

```

在上面的代码中，我们通过结合 `html5.h` 所提供的浏览器事件系统和“`EM_ASM_`”宏函数，将事件的发生状态传递到了上层的 JavaScript 环境中，同时根据当前事件发生时鼠标位置的坐标值来实时地更换网页背景颜色。关于鼠标事件结构体“`EmscriptenMouseEvent`”的详细属性值，可以参考官方文档中的相关说明。

触摸事件 (Touch)

主要的 C/C++ 源代码如下：

```

html5_touch.cc
#include <emscripten/html5.h>
#include <emscripten.h>
#include <vector>
#include <cmath>
// 定义一个触摸点在消息队列中的默认索引位置
#define DEFAULT_POINR_INDEX 0

using namespace std;

```



```

// 用于存放初始的拖曳位置
static vector<long> START_POINT;
// 用于计算拖曳距离的函数
double calMoveDistance (long x, long y) {
    return sqrt(abs(y * y - x * x));
}
// 触摸开始事件的回调函数
EM_BOOL em_touchstart_event_callback (
    int eventType,
    const EmscriptenTouchEvent *e,
    void *userData
) {
    // 保存起始点的触摸位置
    START_POINT = vector<long>{
        e->touches[DEFAULT_POINR_INDEX].screenX,
        e->touches[DEFAULT_POINR_INDEX].screenY
    };
    return EM_TRUE;
}
// 触摸结束事件的回调函数
EM_BOOL em_touchend_event_callback (
    int eventType,
    const EmscriptenTouchEvent *e,
    void *userData
) {
    long x = e->touches[DEFAULT_POINR_INDEX].screenX;
    long y = e->touches[DEFAULT_POINR_INDEX].screenY;
    // 计算移动距离
    double distance = calMoveDistance(y - START_POINT[1], x - START_POINT[0]);

    EM_ASM({
        // 打印移动距离结果
        console.log(`Drag for "${0.toFixed(2)}px"`);
    }, distance);

    return EM_TRUE;
}

int main (int argc, char **argv) {

```

```
// 用于注册触摸开始事件的函数
emscripten_set_touchstart_callback(
    "#window",
    nullptr,
    EM_FALSE,
    &em_touchstart_event_callback);
// 用于注册触摸结束事件的函数
emscripten_set_touchend_callback(
    "#window",
    nullptr,
    EM_FALSE,
    &em_touchend_event_callback);

return 0;
}
```

上面我们构建了一个稍微复杂一点的 Wasm 应用，该应用主要通过监听浏览器的 `touchstart` 和 `touchend` 两个事件，间接地计算一次完整的触摸事件在页面上的移动距离（绝对距离）。整个应用的构建思路十分简单：首先通过监听 `touchstart` 事件，保存触摸事件初次发生时的触摸点位置信息（即向量“`START_POINT`”中的数据），然后在“`touchend`”事件发生时，通过结合当前触摸点的位置信息与之前保存的初始触摸点位置信息，即可得到整个触摸事件在页面上的绝对移动距离（以像素为单位）。

7.4 基于 EGL、OpenGL、SDL 和 OpenAL 的多媒体处理

Emscripten 在其内部整合并支持了众多用于构建复杂 2D/3D 多媒体应用的行业标准规范与相应的 API 库。本节将主要介绍基于 EGL 的传统 OpenGL、OpenGL-ES 以及 OpenAL 等图形与音频处理库来构建上层 Wasm 应用的基本方法和相关示例。

EGL（OpenGL ES Native Platform Graphics Interface）是由非营利性组织 Khronos Group 于 2000 年创建的一套用于连接上层图形渲染 API（如传统 OpenGL、OpenGL-ES 和 OpenVG）与本地操作系统平台的接口规范。EGL 可用于管理图形上下文的创建、渲染同步等与平台视窗系统相关的功能。但实际上，EGL 在常见的操作系统及图形驱动程序中并没有被广泛使用，其最吸引人的地方则是：在开发 Android 应用时，EGL 被作为在使用 Android NDK 为 OpenGL-ES 创建渲染上下文时的主要接口规范。另外，基于 MIT 协议的著名开源三维计算机图形库“Mesa”，也在其图形驱动程序中实现了 EGL 规范。

通过基于 EGL 构建的视窗系统,我们可以使用 OpenGL 的相关 API 来向该视窗中绘制内容。Emscripten 在其内部提供了多种策略用以兼容 OpenGL 接口在 Web 平台上的调用和执行。在默认情况下, Emscripten 推荐在构建原生应用时直接使用对 WebGL 友好的 OpenGL-ES 2.0 子集, 该子集中的接口将被直接映射到浏览器平台中 WebGL 所对应的具有相同功能的 GL-ES 命令集上。从另一个方面来讲, 这种从 OpenGL 到 WebGL 的直接映射过程, 也会在最大程度上保留原始应用在使用 OpenGL 绘制图形时所具有的高性能。

7.4.1 使用 EGL 与 OpenGL 处理图形

在本节内容中, 我们将分别介绍如何基于 EGL 与 OpenGL 来构建简单的 Wasm 应用。在构建应用时, 我们将使用这些图形处理库对应的原生 C/C++ API, Emscripten 需要做的只是通过一系列的关系绑定过程将这些 API 的运行环境从原生平台迁移到 Web 浏览器平台上。比如对于一部分原先在 C/C++ 代码中使用的 OpenGL 接口, 当它们被迁移到 Web 平台上执行时, 将会被具有相同功能的 WebGL 接口所代替。

EGL

EGL 的原生 API 是基于 C 语言开发的, 示例代码如下:

```
egl.cc
#include <iostream>
#include <EGL/egl.h>

using namespace std;

int main(int argc, char *argv[]) {
    // 获取显示器
    EGLDisplay display = eglGetDisplay(EGL_DEFAULT_DISPLAY);
    EGLint major, minor;
    // 初始化 EGL
    eglInitialize(display, &major, &minor);

    EGLint numConfigs;

    eglGetConfigs(display, NULL, 0, &numConfigs);

    EGLint attribs[] = {
        EGL_RED_SIZE, 5,
```

```

    EGL_GREEN_SIZE, 6,
    EGL_BLUE_SIZE, 5,
    EGL_NONE
};
EGLConfig config;
// 选择配置
eglChooseConfig(display, attribs, &config, 1, &numConfigs);

EGLNativeWindowType window;
// 构造绘制表面:
EGLSurface surface = eglCreateWindowSurface(display, config, window, NULL);

EGLint width, height;
// 查询该“绘图表面”的相关信息
eglQuerySurface(display, surface, EGL_WIDTH, &width);
eglQuerySurface(display, surface, EGL_HEIGHT, &height);
// 打印信息
cout << "Screen width(px): " << width << endl;
cout << "Screen height(px): " << height << endl;

// 通过 OpenGL API 绘制画面
...
}

```

在这段代码中，我们通过 EGL 的原生 API 获取到了当前构建 Wasm 应用所默认使用的“绘图表面”，即 Emscripten 的默认“绘图表面”其长宽信息，并同时将这些信息打印到了浏览器控制台中。由于 Emscripten 运行时环境已经默认整合了 EGL 库函数的编译时绑定，因此可以直接使用如下命令来编译该应用。

```

emcc egl.cc
-o egl.html
-s WASM=1

```

由于 Emscripten 在其内部并没有完全实现 EGL 1.4 版本中的所有库函数，因此在实际构建和编译 EGL 应用时，应注意潜在的兼容性问题。

OpenGL

在上面基于 EGL 构建的绘图上下文基础上，我们通过调用 OpenGL 的相关 API 在 Emscripten 的默认绘图表面上绘制内容。可以看到，EGL 为 OpenGL 构建了初始的绘图容器，而 OpenGL 则主要负责在这个绘图容器中通过调用 GPU 来绘制实际的内容。示例代码如下：

```
opengl.cc
#include <iostream>
#include <EGL/egl.h>
#include <GLLES2/gl2.h>

// 全局资源
GLuint userProgramObject;

// 创建着色器，加载并编译着色器
GLuint loadShader (GLenum type, const char *shaderSrc) {
    GLuint shader;
    GLint compiled;

    // 创建着色器对象
    shader = glCreateShader(type);

    if (shader == 0)
        return 0;

    // 加载着色器资源
    glShaderSource(shader, 1, &shaderSrc, NULL);

    // 编译着色器
    glCompileShader(shader);

    // 检查编译错误
    glGetShaderiv(shader, GL_COMPILE_STATUS, &compiled);

    if (!compiled) {
        GLint infoLen = 0;
        // 提取错误信息
        glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &infoLen);
        // 删除着色器
        glDeleteShader(shader);
        return 0;
    }
    return shader;
}

// 初始化着色器和应用对象
```

```
int initialize_stage () {
    char vShaderStr[] =
        "attribute vec4 vPosition;    \n"
        "void main()                  \n"
        "{                            \n"
        "    gl_Position = vPosition;    \n"
        "}"                                \n";

    char fShaderStr[] =
        "precision mediump float;      \n"
        "void main()                    \n"
        "{                            \n"
        "    gl_FragColor = vec4 ( 1.0, 0.0, 0.0, 1.0 );\n"
        "}"                                \n";

    GLuint vertexShader;
    GLuint fragmentShader;
    GLuint programObject;
    GLint linked;

    // 加载顶点和片段着色器
    vertexShader = loadShader (GL_VERTEX_SHADER, vShaderStr);
    fragmentShader = loadShader (GL_FRAGMENT_SHADER, fShaderStr);

    // 创建程序对象
    programObject = glCreateProgram();

    if (programObject == 0)
        return 0;
    // 将着色器附加到程序对象上
    glAttachShader(programObject, vertexShader);
    glAttachShader(programObject, fragmentShader);

    // 绑定属性
    glBindAttribLocation(programObject, 0, "vPosition");

    // 链接程序
    glLinkProgram(programObject);

    // 检查链接状态（同上）
```

```
glGetProgramiv(programObject, GL_LINK_STATUS, &linked);

if (!linked) {
    GLint infoLen = 0;

    glGetProgramiv(programObject, GL_INFO_LOG_LENGTH, &infoLen);
    glDeleteProgram(programObject);
    return GL_FALSE;
}

// 存储程序对象
userProgramObject = programObject;

glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
return GL_TRUE;
}

// 主绘制函数
void draw (int winWidth, int winHeight) {
    GLfloat vVertices[] = { 0.0f, 0.5f, 0.0f,
                           -0.5f, -0.5f, 0.0f,
                           0.5f, -0.5f, 0.0f };

    GLuint vertexPosObject;
    // 创建 VBO
    glGenBuffers(1, &vertexPosObject);
    // 绑定缓冲区对象
    glBindBuffer(GL_ARRAY_BUFFER, vertexPosObject);
    // 将顶点数据传递给 VBO
    glBufferData(GL_ARRAY_BUFFER, 9*4, vVertices, GL_STATIC_DRAW);

    // 设置视图
    glViewport(0, 0, winWidth, winHeight);

    glClear(GL_COLOR_BUFFER_BIT);
    glUseProgram(userProgramObject);

    // 加载顶点数据
    glBindBuffer(GL_ARRAY_BUFFER, vertexPosObject);
    glVertexAttribPointer(0, 3, GL_FLOAT, 0, 0, 0);
    glEnableVertexAttribArray(0);
```

```
    glDrawArrays(GL_TRIANGLES, 0, 3);
}

int main(int argc, char *argv[]) {
    // 与 EGL 相关的流程（注释省略）
    EGLDisplay display = eglGetDisplay(EGL_DEFAULT_DISPLAY);

    EGLint major, minor;
    eglInitialize(display, &major, &minor);

    eglBindAPI(EGL_OPENGL_ES_API);

    EGLint numConfigs;
    eglGetConfigs(display, NULL, 0, &numConfigs);

    EGLint attribs[] = {
        EGL_RED_SIZE, 5,
        EGL_GREEN_SIZE, 6,
        EGL_BLUE_SIZE, 5,
        EGL_NONE
    };

    EGLConfig config;
    eglChooseConfig(display, attribs, &config, 1, &numConfigs);

    EGLNativeWindowType window;
    EGLint surfaceAttributes[] = { EGL_NONE };
    EGLSurface surface = eglCreateWindowSurface(display, config, window,
    surfaceAttributes);

    EGLint width, height;
    eglQuerySurface(display, surface, EGL_WIDTH, &width);
    eglQuerySurface(display, surface, EGL_HEIGHT, &height);

    EGLint contextAttributes[] = { EGL_CONTEXT_CLIENT_VERSION, 2, EGL_NONE };
    // 创建 EGL 上下文
    EGLContext context = eglCreateContext(display, config, NULL, contextAttributes);

    eglMakeCurrent(display, surface, surface, context);
```



```
// 初始化 OpenGL 相关资源
initialize_stage();
// 绘制多边形
draw(width, height);
eglSwapBuffers(display, surface);

return 0;
}
```

在上面的代码中，我们通过结合使用 EGL 和 OpenGL-ES 的相关 API 接口，在 Emscripten 的默认绘图容器（Canvas 画布）中绘制了一个红色的三角形。该 Wasm 应用的最终运行效果如图 7-24 所示。

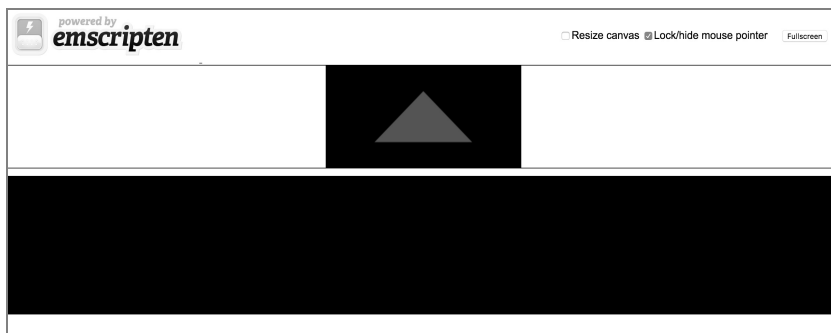


图7-24 上述Wasm应用的最终运行效果

需要注意的是，在编译该应用时，需要在编译命令中添加一个额外的“-s USE_WEBGL2=1”参数。通过该参数，可以让 Emscripten 在编译生成的 JavaScript“胶水”脚本文件中保留与 OpenGL 相关的 API 接口绑定函数，这些函数能够在应用运行时将 C/C++代码中调用的 OpenGL-ES 接口直接映射到对应的 WebGL 接口上。

7.4.2 使用 SDL 处理图形

前面我们介绍过，Emscripten 为 Web 浏览器实现了一个与 SDL 基本完全兼容的 API 绑定环境。因此，可以在基本不需要修改代码的情况下，直接通过 emcc 编译器来编译和移植现有的原生 SDL 应用。示例代码如下：

```
sdl.cc
#include <SDL.h>
#include <stdio>
```

```
// 声明全局的窗体对象
SDL_Window* gWindow = NULL;

// 声明该窗体对象的绘图表面
SDL_Surface* gScreenSurface = NULL;

// 声明将被渲染到绘图表面的图片资源
SDL_Surface* picHandler = NULL;

bool init() {
    bool success = true;

    // 初始化 SDL
    if (SDL_Init(SDL_INIT_VIDEO) < 0) {
        printf("SDL could not initialize! SDL_Error: %s\n", SDL_GetError());
        success = false;
    } else {
        // 创建窗体对象
        gWindow = SDL_CreateWindow(
            "SDL Tutorial",
            SDL_WINDOWPOS_UNDEFINED,
            SDL_WINDOWPOS_UNDEFINED,
            640,
            480,
            SDL_WINDOW_SHOWN);
        if (gWindow == NULL) {
            printf("Window could not be created! SDL_Error: %s\n", SDL_GetError());
            success = false;
        } else {
            // 获得窗体绘图表面
            gScreenSurface = SDL_GetWindowSurface(gWindow);
        }
    }

    return success;
}

bool loadMedia() {
```

```
bool success = true;

// 加载图片资源
picHandler = SDL_LoadBMP("/hello_sdl.bmp");
if (picHandler == NULL) {
    printf("Unable to load image %s! SDL Error: %s\n", "/hello_sdl.bmp", SDL_GetError());
    success = false;
}
return success;
}

// 垃圾收集
void close() {
    // 回收绘图表面对象
    SDL_FreeSurface(picHandler);
    picHandler = NULL;

    // 销毁窗体对象
    SDL_DestroyWindow(gWindow);
    gWindow = NULL;

    // 退出 SDL 子系统
    SDL_Quit();
}

int main(int argc, char* args[]) {
    if (!init()) {
        printf("Failed to initialize!\n");
    } else {
        if (!loadMedia()) {
            printf("Failed to load media!\n");
        } else {
            // 将图片资源绘制在窗体表面上
            SDL_BlitSurface(picHandler, NULL, gScreenSurface, NULL);
            // 刷新视图
            SDL_UpdateWindowSurface(gWindow);
        }
    }
    close();
}
```

```
return 0;
}
```

上述 SDL 应用在 Web 平台上的实际运行效果如图 7-25 所示。

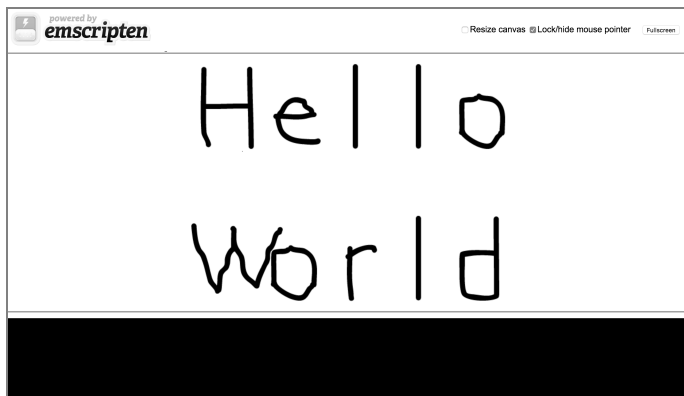


图7-25 上述SDL应用在Web平台上的实际运行效果

与基于 Emscripten 编译原生 OpenGL 应用的过程类似，这里在编译 SDL 应用时，也需要为 emcc 另外指定一个特殊的“USE_SDL=2”参数。通过该参数，可以让 Emscripten 在编译“胶水”脚本文件时保留与 SDL 相关的 JavaScript 绑定代码。不仅如此，Emscripten 还在其内部实现了对 SDL2_image、SDL2_net 等常用 SDL 库的移植绑定过程。读者可以通过执行“emcc -show-ports”命令，来查看这些移植库对应的在使用 emcc 编译时需要添加的额外参数。

7.4.3 使用 OpenAL 处理音频

借助 Emscripten 工具链，不仅可以基于 OpenGL 构建的原生图形应用直接移植到 Web 平台上运行，而且还可以将基于 OpenAL 开放音效标准构建的原生音频应用也无痛地移植到浏览器环境中运行。根据 OpenAL 1.1 标准，Emscripten 在其内部实现了自己的一套 API，这些 API 将基于浏览器平台中现有的 Web Audio 体系来播放和控制音效。示例代码如下：

```
openal.cc

#include <stdio>
#include <stdlib>
#include <stdint>
#include <AL/al.h>
#include <AL/alc.h>

int main(int argc, char **argv) {
```

```
int major, minor;
// 获取 ALC 版本号
alcGetIntegerv(NULL, ALC_MAJOR_VERSION, 1, &major);
alcGetIntegerv(NULL, ALC_MINOR_VERSION, 1, &minor);

printf("ALC version: %i.%i\n", major, minor);
printf("Default device: %s\n", alcGetString(NULL, ALC_DEFAULT_DEVICE_SPECIFIER));
// 打开设备
ALCdevice* device = alcOpenDevice(NULL);
ALCcontext* context = alcCreateContext(device, NULL);
alcMakeContextCurrent(context);
// 获取 OpenAL 版本信息
printf("OpenAL version: %s\n", alGetString(AL_VERSION));
printf("OpenAL vendor: %s\n", alGetString(AL_VENDOR));
printf("OpenAL renderer: %s\n", alGetString(AL_RENDERER));

ALuint buffers[1];

alGenBuffers(1, buffers);
// 打开音频文件
FILE* source = fopen("audio.wav", "rb");
fseek(source, 0, SEEK_END);
int size = ftell(source);
fseek(source, 0, SEEK_SET);
// 读入媒体文件内容
unsigned char* buffer = (unsigned char*) malloc(size);
fread(buffer, size, 1, source);
// 设置偏移量, 忽略头部标识数据
unsigned offset = 12;
offset += 8;
offset += 2;
// 获取采样频率信息
unsigned channels = buffer[offset + 1] << 8;
channels |= buffer[offset];
offset += 2;
printf("Channels: %u\n", channels);
// 获取频率信息
unsigned frequency = buffer[offset + 3] << 24;
frequency |= buffer[offset + 2] << 16;
```

```
frequency |= buffer[offset + 1] << 8;
frequency |= buffer[offset];
offset += 4;
printf("Frequency: %u\n", frequency);

offset += 6;
// 获取 Bits 解析度信息
unsigned bits = buffer[offset + 1] << 8;
bits |= buffer[offset];
offset += 2;
printf("Bits: %u\n", bits);
// 获取声音格式
ALenum format = 0;
if (bits == 8) {
    if (channels == 1)
        format = AL_FORMAT_MONO8;
    else if (channels == 2)
        format = AL_FORMAT_STEREO8;
} else if (bits == 16) {
    if (channels == 1)
        format = AL_FORMAT_MONO16;
    else if (channels == 2)
        format = AL_FORMAT_STEREO16;
}

offset += 8;

printf("Start offset: %d\n", offset);
// 将媒体数据写入缓冲区中
alBufferData(buffers[0], format, &buffer[offset], size - offset, frequency);

ALuint sources[1];
// 创建一个声源
alGenSources(1, sources);
// 设置声源的整型值
alSourcei(sources[0], AL_BUFFER, buffers[0]);
// 播放声音
alSourcePlay(sources[0]);
```

```
    return 0;  
}
```

该应用会从 Emscripten 虚拟文件系统中读取名为“audio.wav”的音频文件，并通过调用与 OpenAL 相关的 API 来分析该音频文件的相关属性（采样频率及解析度等），最后 Emscripten 会在编译生成的“胶水”脚本中间接地通过 Web Audio 来播放该音频。因此，这里在编译该应用时，需要在命令中加入“--preload-file”或“--embed-file”参数，提前将音频文件初始化到 Emscripten 运行时环境的虚拟文件系统中。

7.5 调试 WebAssembly 应用

当我们基于 Emscripten 编写好一个完整的 WebAssembly 应用对应的 C/C++ 及 JavaScript 代码后，经常会在其编译或实际运行阶段遇到各种异常问题，进而导致应用无法正常运行。不仅如此，由于基于 Emscripten 构建的 Wasm 应用本身又具有比原生应用更为复杂的代码逻辑，因此，调试应用并排查问题的过程将变得十分艰难。为此，Emscripten 工具链为我们整合了多种调试手段，以用于分析并解决应用本身存在的异常问题。

7.5.1 编译器的调试信息

通过在编译命令中添加“编译器调试信息标志位”对应的参数，Emscripten 可以在所编译的代码中保留调试信息，甚至创建从源代码到编译代码的直接映射（Source Map）关系。这样便可帮助我们在浏览器中以“单步执行代码（步进）”的方式来调试应用对应的 C/C++ 源代码。

在默认情况下，Emscripten 的编译器入口脚本 emcc 在添加了优化参数（如 -O1、-O2、-Oz 等）的编译流程中，会删除所生成 JavaScript “胶水”代码，以及编译中间代码中的大部分调试信息。比如对于“-O1”及以上级别的优化参数，emcc 会移除源代码对应 LLVM-IR 中间代码中的调试信息，并同时禁用运行时的断言（Assertion）检查。而从“-O2”优化级别开始，JavaScript “胶水”脚本文件将会通过 GCC 来做进一步的优化压缩处理，从而失去了其可读性。

为了能够方便地对应用进行调试，Emscripten 提供了“-g[n]”参数用于保留编译过程中产生的调试信息。在默认情况下，参数“-g”将保留源代码中的空格符号、各函数及变量的名称。不仅如此，我们还可以通过更改该参数方括号中的数字“n”，来改变保留和输出调试信息的等级。该数字可以被设置为 0~4 共 5 个等级，其中每一个等级都会在前一个等级的基础上保留和输出更多的调试信息。值得一提的是，参数“-g4”所对应的调试等级能够生成从应用 C/C++ 源代码到 Wasm 可读代码的直接映射关系。这使得我们可以直接在浏览器中调试编译前的 C/C++

源代码，将发生异常的代码信息更直观地展现出来。

比如对于下面的示例应用，首先编译该应用，通过添加“-g4”参数来保留调试信息，并生成源代码与原始 C/C++代码的映射关系。最后再通过浏览器直接调试该应用的 C/C++源代码。

```
debug.cc
#include <iostream>
#include <emscripten.h>

using namespace std;

extern "C" char EMSCRIPTEN_KEEPALIVE add (char x, char y) {
    cout << "This comes from C++." << endl;
    return x + y;
}
```

在上层 JavaScript 代码中调用从模块中暴露出的方法。

```
post-script.js
__ATPOSTRUN__.push(() => {
    console.log(Module['_add'](10, 20));
});
```

现在我们通过如下命令语句来编译该应用。

```
emcc debug.cc
-s WASM=1
-o debug.html
-g4
--post-js post-script.js
--source-map-base "http://localhost:8888/"
```

这里在编译命令中不仅添加了“-g4”参数，还另外添加了名为“--source-map-base”的参数。该参数主要用于指定 Source Map 文件的所在域地址。比如这里指定的 `http://localhost:8888/`即为该文件的本地域地址。通过该参数，emcc 能够将其编译生成的 Source Map 文件，以“<域地址>+<文件名>+.map”的 URL 形式嵌入到 Wasm 模块内的 `sourceMappingURL` 自定义段结构中。根据 Post-MVP 标准中的相关规范，该段结构将用于浏览器识别及提取应用的 C/C++源代码与 Wasm 模块可读文本代码之间的映射关系。

由于 Source Map 规范属于 WebAssembly Post-MVP 标准中的一部分，因此这里暂时只能通过 Firefox 的 Developer Edition 版本浏览器来体验该功能。在 Web 浏览器中调试上述 Wasm 应

用对应 C/C++源代码的效果如图 7-26 所示。

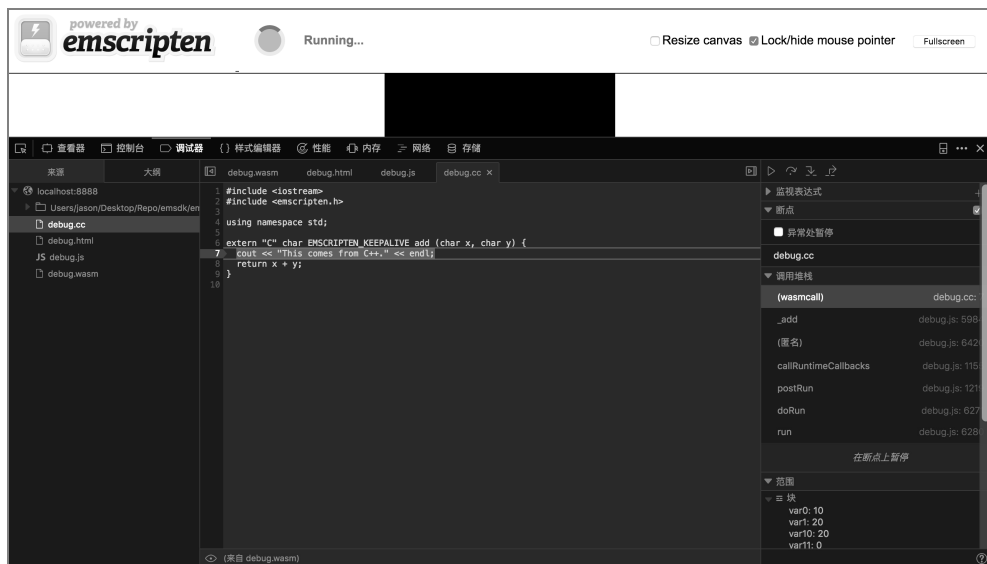


图 7-26 在浏览器中调试Wasm应用对应的C/C++源代码

需要注意的是，当优化参数与调试参数一起使用时，编译器可能会禁用优化阶段的某些优化策略。因此，在编译即将要发布到线上的应用时，请确保在编译命令中没有加入任何与调试相关的参数，以使得编译器的优化过程能够正常进行。

7.5.2 使用调试模式

除通过设置“-g[n]”参数来保留与调试相关的信息外，也可以通过设置“EMCC_DEBUG”全局环境变量，来让 emcc 在编译应用时启用 Emscripten 的调试模式。

如图 7-27 所示，通过使用“EMCC_DEBUG=1”参数，可以让编译器在编译应用源代码时，输出整个编译流程中各关键节点的状态信息。不仅如此，编译器（emcc）还会在编程过程中生成各个编译阶段的中间代码文件，这些文件将被存放在“[TEMP_DIR]”路径下、以“emscripten_temp_*_archive_contents”形式命名的各个文件夹内。其中“[TEMP_DIR]”表示定义在全局配置文件“.emscripten”中的同名环境变量值（在前几章内容中介绍过）。

除此之外，还可以将上述参数更改为“EMCC_DEBUG=2”。此时编译器会将在 JavaScript 优化处理阶段产生的每一个中间代码文件也同样输出到对应的“[TEMP_DIR]”路径下，以便于应用调试。

```

➔ debug git:(master) X EMCC_DEBUG=1 sudo emcc debug.cc -s WASM=1 -o debug.html --post-js post-script.js
Password:
DEBUG:root:PYTHON not defined in /Users/jason/.emscripten, using "/Library/Frameworks/Python.framework/Vers
DEBUG:root:JAVA not defined in /Users/jason/.emscripten, using "java"
DEBUG:root:Cache: PID 37461 acquiring multiprocess file lock to Emscripten cache at /Users/jason/.emscripten
DEBUG:root:Cache: done
DEBUG:root:Cache: PID 37461 released multiprocess file lock to Emscripten cache at /Users/jason/.emscripten
DEBUG:root:check tells us to use asm.js backend
WARNING:root:invocation: /Users/jason/Desktop/Repo/emsdk/emscripten/1.38.0/emcc debug.cc -s WASM=1 -o debug.
bAssembly/Emscripten/debug)
DEBUG:root:Checking JS engine [' /Users/jason/Desktop/Repo/emsdk/node/8.9.1_64bit/bin/node']
INFO:root:(Emscripten: Running sanity checks)
DEBUG:root:compiling to bitcode
DEBUG:root:emcc step "parse arguments and setup" took 0.64 seconds
DEBUG:root:compiling source file: debug.cc
DEBUG:root:running: /Users/jason/Desktop/Repo/emsdk/clang/e1.38.0_64bit/clang++ -target asmjs-unknown-emscri
ny__=0 -D_LIBCPP_ABI_VERSION=2 -Werror=implicit-function-declaration -nostdinc -Xclang -nobuiltininc -Xclang
en/1.38.0/system/include/libcxx -Xclang -isystem/Users/jason/Desktop/Repo/emsdk/emscripten/1.38.0/system/lib
pten/1.38.0/system/include/compat -Xclang -isystem/Users/jason/Desktop/Repo/emsdk/emscripten/1.38.0/system/i
system/include/SSE -Xclang -isystem/Users/jason/Desktop/Repo/emsdk/emscripten/1.38.0/system/include/libc -Xc
/libc/musl/arch/emscripten -Xclang -isystem/Users/jason/Desktop/Repo/emsdk/emscripten/1.38.0/system/local/in
ers/jason/Desktop/Repo/emsdk/emscripten/1.38.0/system/include/SDL -emit-llvm -c -o /tmp/tmpFP6Wep/debug_0.0
DEBUG:root:emcc step "bitcodeize inputs" took 0.58 seconds
DEBUG:root:emcc step "process inputs" took 0.00 seconds
DEBUG:root:will generate JavaScript
DEBUG:root:binaryen root already set to /Users/jason/Desktop/Repo/emsdk/clang/e1.38.0_64bit/binaryen

```

图7-27 编译器在编译应用源代码时输出的调试信息

7.5.3 手动跟踪

除可以通过上面介绍的两种 Emscripten 编译器自带的调试信息输出方式来进行应用调试以外，我们还可以使用一种最基本、最常见的代码调试方式——通过在源代码中添加打印语句来输出各个目标变量的实际值，然后通过比较目标值与实际值是否相等来判断应用产生异常的代码位置。下面将分别针对 C/C++ 与 JavaScript 源代码进行介绍。

C/C++

对于传统的 C/C++ 源代码，我们可以直接使用 `cout` 对象或 `printf` 方法来打印信息，并在上层环境中输出目标变量的值。示例代码如下：

```

#include <iostream>
#include <vector>
#include <map>
#include <string>
#include <emscripten.h>

using namespace std;

extern "C" char EMSCRIPTEN_KEEPALIVE add (char x, char y) {
    // 字符串

```

```

string s = "This comes from C++.";
cout << s << endl;

// 向量
vector<int> v{1, 2, 3, 4, 5};
for (auto i : v) {
    cout << i << endl;
}

// 字典
map<int, int> m = {
    pair<int, int> (1, 2),
    pair<int, int> (2, 3)
};
for (auto i : m) {
    cout << i.first << ":" << i.second << endl;
}

// 基本类型
int d = 10;
double f = 1.5;
cout << d << " " << f << endl;

return x + y;
}

```

JavaScript

对于 JavaScript 源代码，我们也可以通过类似的方式来打印目标变量的实际值，并跟踪某一位置代码的实际调用链路（调用栈）。示例代码如下：

```

__ATPOSTRUN__.push(() => {
    // 调用被 Name Mangling 影响的导出函数
    console.log(Module['__Z3addcc'](10, 20));
    // 打印当前位置的调用堆栈
    console.log(jsStackTrace());
    // 查看原始函数签名
    console.log(demangleAll("__Z3addcc")); // "__Z3addcc [add(char, char)]"
});

```

可以看到，这里首先使用了 Emscripten 运行时环境提供的 jsStackTrace 函数来跟踪源代码

中某一位置代码的实际调用链路。该函数的代码实现细节如图 7-28 所示。其本质也是通过封装并抛出“Error”对象的方式，来检查当前调用位置在整个源代码中的实际调用链路。

```
emscripten/src/preamble.js
Lines 817 to 832 in bf0bdc5

817     function jsStackTrace() {
818         var err = new Error();
819         if (!err.stack) {
820             // IE10+ special cases: It does have callstack info, but it is only populated
821             // so try that as a special-case.
822             try {
823                 throw new Error(0);
824             } catch(e) {
825                 err = e;
826             }
827             if (!err.stack) {
828                 return '(no stack trace available)';
829             }
830         }
831         return err.stack.toString();
832     }
```

图7-28 jsStackTrace函数的代码实现细节

接下来，我们还使用了在 Emscripten 运行时环境中定义的 `demangleAll` 函数，来查看一个被 Name Mangling 影响的导出函数其实际函数定义。需要注意的是，如果想要在全局环境或代码中使用该函数，则需要在编译源代码时，通过添加“-s DEMANGLE_SUPPORT=1”参数来让 `emcc` 在“胶水”脚本文件中保留该函数对应的代码实现。

7.5.4 其他常用编译器调试选项

除了前面介绍的几个专门用于进行代码调试的 `emcc` 编译器参数之外，下面再介绍另外两个与代码调试相关的常用参数。

ASSERTIONS=1

该参数用于启用 `emcc` 对常见内存分配错误（如：写入的内存量多于实际分配的内存量）的运行时断言检查和错误处理机制。我们可以将该参数值更改为“2”以运行更多的附加测试。在没有添加任何优化参数的情况下，`emcc` 会默认按照值为“1”时的调试状态开启该参数。

SAFE_HEAP=1

该参数用于为编译过程添加额外的内存访问检查，并对“内存对齐”和“空指针解引用”给出了详细的错误原因。

关于 `emcc` 所支持参数的类型和具体使用方法，可以参考 `emsdk/emscripten/<version>/src/settings.js` 文件中对应注释给出的详细说明。

第 8 章

WebAssembly 综合实践、发展与未来

在前几章的内容中，我们从整体上介绍了基于 Emscripten 工具链构建一个 WebAssembly 应用的基本流程，以及从编写 C/C++ 源代码到绑定语言关系、使用虚拟文件系统、使用 HTML 5 事件系统、调试应用等各个环节的相关细节信息。在本章的内容中，我们将从零开始构建一个具有实际应用价值的 WebAssembly 应用。需要说明的是，整个 Wasm 应用的构建过程将基于 Emscripten 工具链来进行，但是由于笔者 emcc 编译器的兼容性问题，在编写上层 JavaScript 代码时将仅使用 ECMAScript 5 标准中规定的相关语法特性。（注：当前版本 emcc 在压缩上层 JavaScript 代码时所采用的 Uglify 工具暂时并不支持 ES 6 的语法特性。）

8.1 DIP 综合实践应用

一直以来，对数据的加/解密、多媒体资源的特效处理等都是十分常见的 CPU 密集计算型任务，即：这些任务在执行时没有过多的逻辑判断过程，大部分时间都是在高效地利用 CPU/GPU 进行基本的数学运算。因此，对于这一类应用，便非常适合 WebAssembly 来大展身手，最大程度地优化任务的执行效率。

8.1.1 应用描述

本节将要构建的是一个 DIP（Digital Image Processing，数字图像处理）类型的应用。在平日里我们能够接触到很多与 DIP 相关的应用软件，比如专门用于绘制生成或处理数字图像的 PhotoShop 程序，以及专业化的图形视频处理软件 Adobe Effect 等。虽然它们各自的功能侧重点不同，但从整体上看均属于 DIP 应用的某种特定类型。

我们先来看一下即将构建的 DIP 应用在浏览器中的实际运行效果，如图 8-1 所示。



图8-1 即将构建的DIP应用在浏览器中的实际运行效果

可以看到，整个应用的运行界面被分为三个部分。其中位于最上方的长方形区域主要用来呈现一段被循环播放的视频画面；中间区域偏下方的位置主要用于显示当前视频的实时播放帧率；位于最下方区域的则是该应用与用户的“交互接口”部分。用户可以通过修改此处所列出的一系列选项，然后点击“确认”按钮，来改变当前应用渲染视频画面的内部实现方式。

从宏观层面来看，该应用的主要目的是为一段特定的视频画面添加滤镜，并对经过滤镜处理后的视频画面进行实时渲染播放。在正式编写代码之前，我们需要先了解为视频画面或静态图像添加滤镜的基本原理，即数字图像处理领域中的一个最基本概念：卷积。

8.1.2 滤镜与卷积

从直观的角度来看，卷积就是通过一个固定大小的方阵（卷积核）来对图像中每一个点的像素值进行相关计算。通常，我们也将对图像中各像素点进行卷积处理的过程称为滤波。在进行滤波时，有两种常见的卷积核类型可以选择。这里将使用名为“相关核”的卷积核类型，通过该核进行滤波时不需要对核矩阵本身进行翻转。

下面我们将通过一个十分简单和直观的例子来进一步了解对数字图像进行滤波处理的基本流程。首先需要设置一个 3×3 规格的方阵作为整个卷积操作中的卷积核矩阵，如图 8-2 所示。

1	2	1
2	1	2
1	2	1

图8-2 DIP卷积核矩阵

为了便于直观地展示流程，如图 8-3 所示，这里给出的目标图像是一个 5×5 规格的灰度图（即没有 RGBA 分量）。该方阵中的每一个数值都代表了图像中对应位置像素点的实际灰阶值。

2	1	5	3	1
1	2	3	5	3
1	2	3	4	1
2	1	3	2	2
1	2	3	4	5

图8-3 灰度图的原始像素矩阵

至此，我们就准备好了所有需要使用的数据信息。接下来便可以根据卷积公式开始进行滤波过程了。

在离散域上进行卷积计算，可以参考如下数学公式来进行。

$$g(i, j) = \sum_{k, l}^n f(i+k, j+l)h(k, l)$$

在该公式中，函数 f 表示目标图像的像素矩阵在 $(i+k, j+l)$ 位置处对应的像素值；函数 h 用于表示卷积核对应方阵在以 (i, j) 为中心基准点时，其相对位置 (k, l) 处的数值。公式左边的 $g(i, j)$ 函数定义了在经过滤波处理后，目标图像的最终像素矩阵在 (i, j) 位置处对应像素值的具体计算过程。接下来，我们将以上面给出的灰度图像素矩阵和卷积核矩阵为例来进行卷积计算。

如果想要计算出在原始像素矩阵中 $(2, 2)$ 位置处的像素点在经过卷积计算后的结果值，那么只需要将该点与卷积核矩阵的中心元素在位置上进行对齐，然后将卷积核矩阵规格范围内的像素矩阵元素值与卷积核元素值两两相乘并进行累加，即可得到最后的像素值。比如这里的 $g(2, 2)$ 便等于多项式 “ $1 * 2 + 2 * 1 + 1 * 5 + 2 * 1 + 1 * 2 + 2 * 3 + 1 * 1 + 2 * 2 + 1 * 3$ ” 的最终值，经过

计算后得到“27”。对于原始像素矩阵中其他类似的像素点，可以依此类推，进行相应的计算。卷积计算的模拟过程如图 8-4 所示。

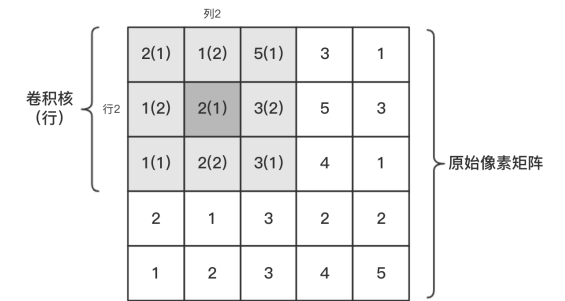


图8-4 卷积计算的模拟过程

但事实上，在原始像素矩阵中，并不是所有的像素点都可以按照上述方式来进行卷积计算的。其中一种特殊情况是：当卷积核矩阵在处理原始像素矩阵中位于图像边缘的像素点时，其范围内的一部分“单元格”可能（1*1 大小的特殊规格卷积核无此问题）无法在原始像素矩阵中找到与之相对应的像素点，那么此时如何计算当前目标像素点的卷积结果值，便直接影响其经过滤波处理后的实际显示效果。在通常情况下，我们会将这种在处理边缘像素点时所遇到的卷积计算问题称为图像处理的边缘效应问题，如图 8-5 所示。

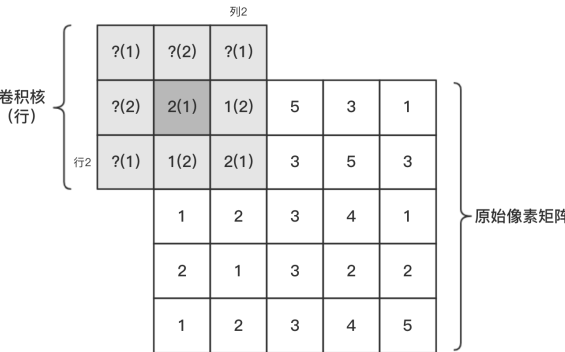


图8-5 在滤波过程中出现的边缘效应问题

从图 8-5 中可以看到，当卷积核矩阵在处理原始像素矩阵中位于左上角的第一个像素点时，整个卷积核中位于第一行、第一列的单元格数据并没有能够与之相乘的原始像素矩阵值。这时，我们便需要通过“边缘像素复制”策略来为这些卷积核单元格分配相对应的原始像素值。如图 8-6 所示，该策略的实现原理十分简单，即通过将原始像素矩阵中位于边缘的像素直接向外侧翻转的方式，为那些没有对应可用像素数据的卷积核单元格分配原始像素数据。

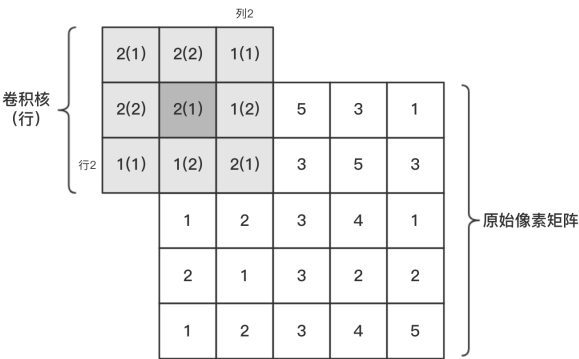


图8-6 “边缘像素复制”策略的实现原理

当然，除使用边缘像素复制策略以外，我们还可以选择直接过滤掉对图像边缘像素的卷积处理过程。这同样是一种十分常用的针对边缘效应问题的解决方法。当整个图像原始像素矩阵中的像素点都通过卷积核（相关核）进行相应的卷积计算后，便得到了如图 8-7 所示的经过滤波处理后的像素矩阵。可以看到，这里在进行卷积操作时，我们选择了直接过滤掉图像边缘像素点这种简单的边缘效应处理方案。

2	1	5	3	1
1	27	93	221	3
1	165	784	2121	1
2	1134	6141	17331	2
1	2	3	4	5

图8-7 上述灰度图经过滤波处理后得到的像素矩阵

从下一节内容开始，我们将根据上面介绍的图像滤镜基本原理来构建对应的 WebAssembly 应用。该应用将基于图 8-8 所示的卷积核矩阵来对视频流中的每一帧图像数据进行相应的卷积处理，并将处理后的像素矩阵数据以图像的形式实时地绘制到页面的指定区域内。

-1	-1	1
-1	14	-1
1	-1	-1

图8-8 即将构建的WebAssembly应用所使用的目标卷积核矩阵

8.1.3 基本组件类型与架构

在开始动手编写代码之前，我们先从整体上给出该 WebAssembly 应用各个组件之间的基本交互逻辑，如图 8-9 所示。

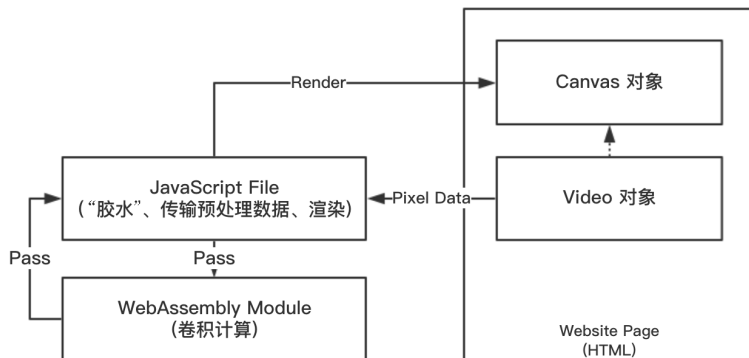


图8-9 该WebAssembly应用各个组件的基本交互逻辑

整个应用从组成结构上可以分为以下三个部分。

UI 交互部分

UI 交互部分为整个 Wasm 应用与用户进行交互的直接入口。根据图 8-1，我们需要在对应的 HTML 文件中放置一个可用于实时渲染视频画面的 Canvas 容器、一个用于显示实时帧率的文字标签，以及一组用于应用交互并控制其底层运行状态的单选按钮和按钮。除此之外，为了能够从视频文件中获取每一帧画面所对应的原始像素矩阵数据，还需要在整个页面中放置一个隐藏的“video”标签来加载视频资源。这样我们便可以将该标签对应的 HTMLVideoElement 图像源绘制到页面上的 Canvas 容器中，然后通过该容器提供的 getImageData 方法来间接地获取对应于视频流每一帧画面的原始像素矩阵数据。

当获取到视频画面的原始像素矩阵数据后，我们便可以在上层代码中分别调用由 C/C++（Wasm）和 JavaScript 语言实现的卷积函数来对矩阵中的像素值进行滤波处理。最后，再通过 Canvas 容器对象的 putImageData 方法，将经过处理后的像素矩阵数据再次绘制到容器中，即可实现视频画面的实时滤镜效果。

核心卷积函数部分

关于核心卷积函数部分的功能，我们会将其分别放在 C/C++ 以及 JavaScript 代码中来实现。其中 C/C++ 代码中的函数实现将会以 Wasm 模块的形式进行封装，并通过相应的上层 Web-API

供该应用进行调用。

“胶水”脚本部分

“胶水”脚本部分的 JavaScript 代码主要用于将编译好的 Wasm 模块、HTML 文件，以及其他具有特定功能的 JavaScript 函数进行连接与整合。除此之外，还需要处理诸如单选钮和按钮的事件绑定与响应、实时渲染的性能检测，以及主渲染循环相关功能的实现。对于主渲染循环，这里将使用 `window.requestAnimationFrame` 函数对渲染在 Canvas 容器上的画面进行实时更新。

接下来，我们将按照上面介绍的应用组成结构来编写其各个组件的具体实现代码。

8.1.4 编写基本页面骨架（HTML 与 CSS）

首先给出的是 UI 交互部分的 HTML 代码实现。

```
index.html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>DEMO-DIP</title>
  <style>
    * {
      font-family: "Arial,sans-serif";
    }
    .fps-num {
      font-size: 50px;
    }
    .video {
      display: none;
    }
    .operation {
      margin: 20px;
    }
    button {
      width: 150px;
      height: 30px;
      margin-top: 10px;
      border: solid 1px #999;
      font-size: 13px;
```

```

        font-weight: bold;
    }
    .radio-text {
        font-size: 13px;
    }
</style>
</head>
<body>
    <canvas class="canvas-scenery"></canvas>
    <div class="operation">
        <h2>帧率: <span class="fps-num">NaN</span> FPS</h2>
        <input name="options" value="0" type="radio" checked="checked"/>
        <span class="radio-text">不开启渲染.<span> <br/>
        <input name="options" value="1" type="radio"/>
        <span class="radio-text">使用 <b>[JavaScript]</b> 渲染.<span> <br/>
        <input name="options" value="2" type="radio"/>
        <span class="radio-text">使用 <b>[WebAssembly]</b> 渲染.<span> <br/>
        <button>确认</button>
    </div>
    <!-- 循环播放视频 -->
    <video class="video" loop="true" autoplay="true" src="media/video.mp4"
type="video/mp4">
    </body>
    <!-- 载入使用 Emscripten 编译生成的“胶水”脚本文件 -->
    <script src="./dip.js"></script>
</html>

```

这部分代码在其本身的实现上并没有太多需要特别介绍的地方。唯一值得注意的是，不要忘记在整段代码的最后，通过“script”标签显式地加载后续将会由 Emscripten 工具链编译生成的 JavaScript “胶水”脚本文件。另外，还要注意应用代码及其他文件资源的整体存放路径。比如这里需要将 video.mp4 视频文件存放到与该 HTML 文件处于同一个目录下的“media”文件夹内。

8.1.5 编写核心卷积函数 (C++)

接下来，我们将要编写使用 C/C++ 代码实现的，用于对图像进行滤波处理的核心卷积函数。

```
dip.cc
```

```

#include <emscripten.h>
#include <cmath>

```

```

/**
 * @brief 核心卷积函数
 * @param {unsigned char*} data - 原始像素矩阵数组
 * @param {int} width - 原始像素矩阵宽度
 * @param {int} height - 原始像素矩阵高度
 * @param {char*} kern - 卷积核矩阵数组
 * @param {int} kWidth - 卷积核矩阵宽度
 * @param {int} kHeight - 卷积核矩阵高度
 * @param {int} divisor - 除法因子
 */
extern "C" void EMSCRIPTEN_KEEPALIVE convFilter (
    unsigned char* data, int width, int height, char* kern,
    int kWidth, kHeight, int divisor
) {
    // 定义一些用于保存 RGB 分量值的变量
    int r, g, b;
    int yy, xx, imageOffset, kernelOffset, pix;
    int kCenterY = floor(kHeight / 2);
    int kCenterX = floor(kWidth / 2);
    // 开始对图像中心区域（除去四角及边缘）的原始像素矩阵进行卷积计算
    for (int y = kCenterY; y < height - kCenterY; ++y) {
        for (int x = kCenterX; x < width - kCenterX; ++x) {
            r = 0;
            g = 0;
            b = 0;
            // 通过卷积核对一个像素点进行卷积计算
            for (int ky = 0; ky < kHeight; ++ky) {
                for (int kx = 0; kx < kWidth; ++kx) {
                    imageOffset = (width * (y - kCenterY + ky) + (x - kCenterX + kx)) * 4;
                    kernelOffset = kWidth * ky + kx;
                    // 对 RGB 颜色中的各分量值分别进行相乘及累加
                    r += data[imageOffset + 0] * kern[kernelOffset];
                    g += data[imageOffset + 1] * kern[kernelOffset];
                    b += data[imageOffset + 2] * kern[kernelOffset];
                }
            }
            pix = (width * y + x) * 4;
            // 使用“除法因子”限制并修正 RGB 各分量的最终取值范围

```

```

    data[pix + 0] = ((r / divisor) > 255) ? 255 : ((r / divisor) < 0) ? 0 : r / divisor;
    data[pix + 1] = ((g / divisor) > 255) ? 255 : ((g / divisor) < 0) ? 0 : g / divisor;
    data[pix + 2] = ((b / divisor) > 255) ? 255 : ((b / divisor) < 0) ? 0 : b / divisor;
  }
}
}

```

在上面代码的编写过程中，需要注意以下几个关键点。

RGBA 分量值

按照应用的设计方案，这里会通过调用 Canvas 容器提供的 `getImageData` 方法，来获取被事先渲染到容器上的视频流原始像素矩阵数据。需要注意的是，不同于我们之前所介绍的针对灰度图的滤波处理过程，这里的 `getImageData` 方法会返回一个包含目标图像像素点色彩信息的一维数组“`Uint8ClampedArray`”。如图 8-10 所示，该数组中的数据将会按照 RGBA 的分量顺序依次循环排列，其中每 4 个数据值对应一个像素点的色彩分量信息。在实际进行卷积计算时，便需要分别针对色彩信息中的 R、G、B 三个分量值同时进行处理。因此，在每一轮卷积过程的循环开始时，我们都需要将当前指向原始像素矩阵数组的指针向后移动“4”个单元，以便让其指向下一个像素对应“R”分量的起始位置。

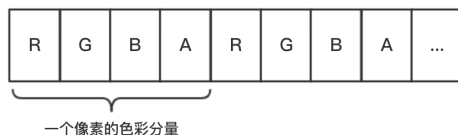


图8-10 通过`getImageData`方法获取像素分量数组

类型对应

由于这里 `Uint8ClampedArray` 数组（`TypedArray`）在其内部存放了类型为“8 位无符号整数”的数字值序列，因此在 C/C++ 代码中则需要使用相对应的 `unsigned char` 类型来声明原始像素矩阵数组内的元素值。同理，其他变量值的具体声明类型，也可以参考其各自的使用场景及取值范围来进行选择。

8.1.6 编写主渲染循环与“胶水”代码（JavaScript）

最后，我们将编写用于整合 WebAssembly 模块、UI 交互层，以及其他与应用本身功能相关的 JavaScript 代码。

```

post-script.js
__ATPOSTRUN__.push(function() {

```

```

var GLOBAL_STATUS = 'STOP';
// 定义监听器，处理页面上的用户交互流程
document.querySelector("button").addEventListener('click', function(e) {
    var aim = Array.prototype.slice.call(document.querySelectorAll("[name='options']"))
    .filter(function(item) {
        if (item.checked) {
            return item;
        }
    });
    var statusFlag = ['STOP', 'JS', 'WASM'];
    // 根据用户的选择（不渲染、使用 JS/Wasm 渲染）来更改相应的状态标志位
    GLOBAL_STATUS = statusFlag[aim.pop().value];
});

// 定义全局变量与引用
var fpsNumDisplayElement = document.querySelector('.fps-num');
// 分别用于存储使用 JavaScript 及 Wasm 版本卷积函数处理每一帧图像数据时的耗时情况
var jsTimeRecords = [], wasmTimeRecords = [];
var clientx, clienty;

// 定义卷积核数组
var kernel = [
    [-1, -1, 1],
    [-1, 14, -1],
    [1, -1, -1]
];
var divisor = 4;

// 基于 JavaScript 实现的核心卷积函数
function jsConvFilter(data, width, height, kernel, divisor) {
    const w = kernel[0].length;
    const h = kernel.length;
    const half = Math.floor(h / 2);
    for (var y = 1; y < height - 1; y += 1) {
        for (var x = 1; x < width - 1; x += 1) {
            const px = (y * width + x) * 4;
            var r = 0, g = 0, b = 0;
            // 核心迭代
            for (var cy = 0; cy < h; ++cy) {
                for (var cx = 0; cx < w; ++cx) {

```

```

        const cpx = ((y + (cy - half)) * width + (x + (cx - half))) * 4;
        r += data[cpx + 0] * kernel[cy][cx];
        g += data[cpx + 1] * kernel[cy][cx];
        b += data[cpx + 2] * kernel[cy][cx];
    }
}

data[px + 0] = ((r / divisor) > 255) ? 255 : ((r / divisor) < 0) ? 0 : r / divisor;
data[px + 1] = ((g / divisor) > 255) ? 255 : ((g / divisor) < 0) ? 0 : g / divisor;
data[px + 2] = ((b / divisor) > 255) ? 255 : ((b / divisor) < 0) ? 0 : b / divisor;
}
}
return data;
}

```

// 用于绑定 Wasm 卷积函数的“胶水”方法

```

function filterWASM (pixelData, width, height) {
    const arLen = pixelData.length;

    // 为原始像素矩阵数据分配共享内存空间（注意使用 Uint8Array）
    const memData = Module['_malloc'](arLen * Uint8Array.BYTES_PER_ELEMENT);

    // 填充数据
    HEAPU8.set(pixelData, memData / Uint8Array.BYTES_PER_ELEMENT);

    // 将卷积核矩阵的二维数组扁平化为一维数组
    const flatKernel = kernel.reduce(function(acc, cur) {
        return acc.concat(cur)
    });

    // 为卷积核矩阵数据分配共享内存空间（注意使用 Int8Array）
    const memKernel = Module['_malloc'](9 * Int8Array.BYTES_PER_ELEMENT);

    // 填充数据
    HEAP8.set(flatKernel, memKernel / Int8Array.BYTES_PER_ELEMENT);

    // 调用 Wasm 版本的核心卷积函数
    Module['_convFilter'](memData, width, height, memKernel, 3, 3, divisor);

    // 从共享线性内存段中获取处理后的数据值

```



```

    const filtered = HEAPU8.subarray(memData / Uint8Array.BYTES_PER_ELEMENT, memData /
    Uint8Array.BYTES_PER_ELEMENT + arLen);

    // 释放内容
    Module['_free'](memData);
    Module['_free'](memKernel);

    // 返回经过处理后的像素矩阵数据
    return filtered;
}

// 包装好的 JavaScript 卷积函数
function filterJS (pixelData, width, height) {
    return jsConvFilter(pixelData, width, height, kernel, divisor);
}

// 用于进行耗时统计的辅助函数
function getAverageTime (vector) {
    // 选取“vector”中最后 AVERAGE_RECORDS_COUNT 条数据记录
    var AVERAGE_RECORDS_COUNT = -20;
    // 计算并返回这些记录的平均值
    return (vector.slice(AVERAGE_RECORDS_COUNT).reduce(function(pre, item) {
        return (pre + item)
    }, 0) / Math.abs(AVERAGE_RECORDS_COUNT)).toFixed(2);
}

// 核心绘制函数
function draw () {
    // 将视频流中的当前一帧画面绘制到 Canvas 容器上
    context.drawImage(video, 0, 0);

    // 获取该帧画面所对应的原始像素矩阵数组
    pixels = context.getImageData(0, 0, video.videoWidth, video.videoHeight);

    // 记录卷积计算过程的开始时间
    const timeStart = performance.now();

    // 根据用户的设置选择对应的处理函数
    if (GLOBAL_STATUS === 'JS') {
        pixels.data.set(filterJS(pixels.data, clientx, clienty));
    }
}

```

```
}
if (GLOBAL_STATUS === 'WASM') {
  pixels.data.set(filterWASM(pixels.data, clientx, clienty));
}

// 计算用时
var timeUsed = Math.round(1000 / (performance.now() - timeStart));

// 将用时记录更新到对应的耗时数组中
if (GLOBAL_STATUS === 'JS') {
  jsTimeRecords.push(timeUsed);
  // 更新 UI 上显示的平均帧率数据
  fpsNumDisplayElement.innerHTML = getAverageTime(jsTimeRecords);
} else if (GLOBAL_STATUS === 'WASM') {
  wasmTimeRecords.push(timeUsed);
  fpsNumDisplayElement.innerHTML = getAverageTime(wasmTimeRecords);
} else {
  fpsNumDisplayElement.innerHTML = 'NaN';
}

// 将经过卷积处理后的像素矩阵数据绘制到 Canvas 容器上
context.putImageData(pixels, 0, 0);

// 启动主循环
requestAnimationFrame(draw);
}

// 获取关键元素的 DOM 引用
var video = document.querySelector('.video');
var canvas = document.querySelector('.canvas-scenery');

// 获取 Canvas 容器的上下文环境
var context = canvas.getContext('2d');

// 视频加载完毕后开始渲染
video.addEventListener("loadeddata", function() {
  canvas.setAttribute('height', video.videoHeight);
  canvas.setAttribute('width', video.videoWidth);

  // 获取舞台大小
  clientx = canvas.clientWidth;
```

```

    clienty = canvas.clientHeight;

    // 开始绘制
    draw(context);
  });
});

```

在编写这部分代码的过程中，需要注意以下几个关键点。

卷积核扁平化

从上面的代码中可以看到，用于描述卷积核矩阵的二维数组被扁平化成对应的一维数组形式，这样可以方便地通过共享线性内存段将其共享到 C/C++ 代码中使用。而卷积核矩阵数组的不同表现形式，也导致我们在编写 JavaScript 和 C/C++ 核心卷积函数时所使用的卷积核数据有着细微差别。在 JavaScript 代码中，可以直接以二维数组 “kernel[cy][cx]” 这种形式来使用对应矩阵某位置上的卷积核数据；而在 C/C++ 代码中，则需要通过 “kernel[kWidth * ky + kx]” 这种线性对应的形式来查找并使用卷积核数据。

帧率统计函数

首先使用了两个名称分别为 “jsTimeRecords” 和 “wasmTimeRecords” 的 JavaScript 数组来收集在某模式（JavaScript / Wasm）下浏览器处理（滤波）每一帧画面数据的耗时情况。然后通过计算对应数组最后 AVERAGE_RECORDS_COUNT 个时间样本的平均值给出该模式的实时 FPS 性能数据。需要注意的是，这里所说的 FPS 其实仅包含了浏览器对原始像素数据的卷积处理时间，并没有包含像素数据的获取以及在 Canvas 容器上进行绘制所产生的耗时。

主循环

所谓的主循环主要有两个功能：一是负责不停地从 Canvas 容器中获取当前视频流的原始画面帧数据；二是负责不停地将经过滤波处理后的画面帧数据实时地绘制到 Canvas 容器上。为了保证绘制的性能，这里使用 requestAnimationFrame 函数来支持整个主循环中的调用过程。在绘制图像时，该函数能够把在一帧时间内发生的所有改动汇聚在一起，然后仅通过一次重绘或回流操作来完成当前画面的状态变更。

8.1.7 使用 Emscripten 编译并运行应用

我们可以使用如下命令来编译该应用。

```

emcc dip.cc
-s WASM=1

```

```
-O3
--post-js post-script.js
-o dip.js
```

最后运行应用。

8.1.8 性能对比

现在，我们将上面构建的 WebAssembly 应用分别运行在最新版本的 Google Chrome 和 Mozilla Firefox 浏览器中，以观察它们对 Wasm 和 JavaScript 两种类型代码的性能优化程度。

- Google Chrome (Version 67.0.3396.99 (Official Build) (64-bit)): 应用的运行结果如图 8-11 所示。

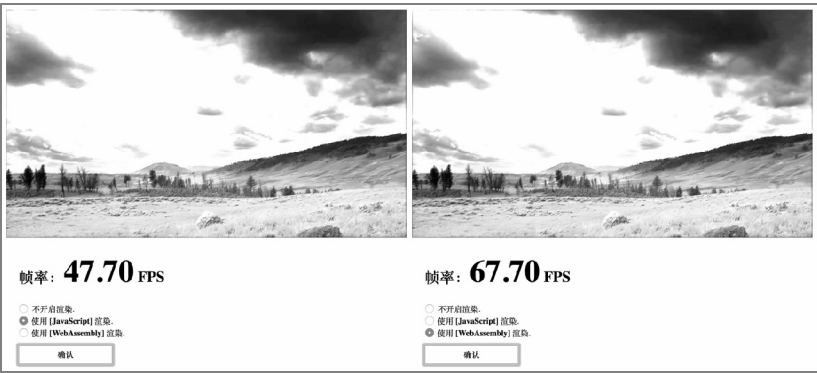


图8-11 在Google Chrome浏览器中应用的运行结果

- Mozilla Firefox (62.0b5 (64-bit)): 应用的运行结果如图 8-12 所示。



图8-12 在Mozilla Firefox浏览器中应用的运行结果

从整体上看,在最新版本的浏览器中,Wasm 应用的运行效率相比半年前已经有了非常明显的提升。对于上述应用,我们看到其性能损耗主要集中在 JavaScript 与 Wasm 环境之间相互传递复杂数据的过程,以及浏览器的执行指针在 JavaScript 与 Wasm 上下文之间的频繁切换上。其中前者主要为一次性的性能损耗,当两者交换的数据体积过大时便会影响应用的整体执行效率;后者则是单次切换所造成的性能损耗,虽然其并不明显,但由于 requestAnimationFrame 函数带来的不断循环及页面重绘,导致轻微的损耗在数量上进行叠加,进而使得应用整体的性能损耗变得十分严重。而这也是 WebAssembly 当前并不十分擅长处理的一种场景。

8.2 WebAssembly 常用工具集

本节我们将介绍现阶段常用的一些与 WebAssembly 相关的本地/在线开发工具、模块调试和转译工具,以及将 Go、Rust 等非 C/C++ 语言代码编译到 Wasm 模块的实现细节。作为在 Wasm 生态中具有举足轻重地位的重要角色,笔者挑选了如下几种常用的工具和特性进行介绍。

8.2.1 Cheerp

如图 8-13 所示,Cheerp 是一个类似于 Emscripten 工具链的,专门用于将基于 C/C++ 语言编写的原生应用平滑地跨平台移植到 Web 浏览器上运行的 WebAssembly / Asm.js 应用开发工具包。虽然其官方给出了十分好听的宣传标语,但作为自信的技术人员,我们可以通过实际的开发流程来体验 Cheerp 和 Emscripten 在构建 Wasm 应用时的不同之处。

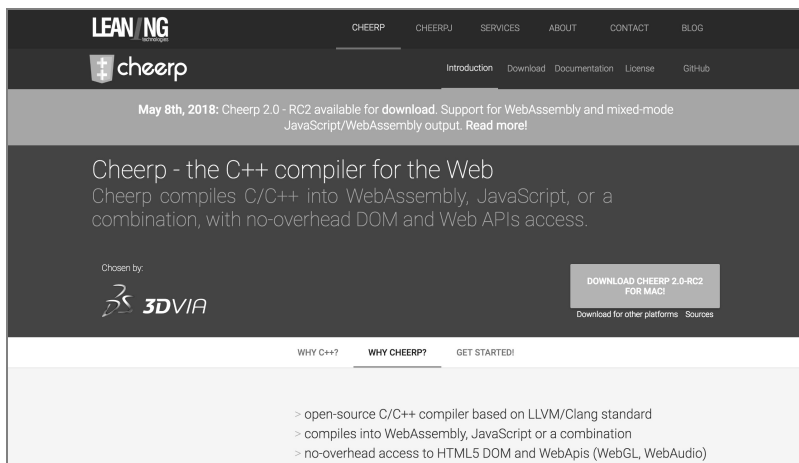


图8-13 Cheerp在其官网给出的介绍

我们按照官方文档给出的示例代码，来基于 Cheerp 构建一个简单的 Wasm 应用。C/C++ 部分的代码如下：

```
hello_cheerp.cc
// 在该头文件中包含了所有对应于浏览器接口的 C/C++ 调用接口
#include <cheerp/clientlib.h>
#include <cheerp/client.h>

// 这里的[[cheerp::genericjs]]标签指示 Cheerp 在 JavaScript 环境中编译该方法
[[cheerp::genericjs]] void domOutput(const char* str) {
    // 这里调用了定义在 client 命名空间中的 console.log 方法，该方法直接对应于浏览器中原生的同名方法
    client::console.log(str);
}

// 这里的 webMain 方法为所有基于 Cheerp 构建的 Web 应用的运行入口
void webMain() {
    // 调用上面定义的 domOutput 方法;
    domOutput("Hello, world!");
}
```

当代码编写完成后，我们可以通过从 Cheerp 官网下载的 MacOS 专用开发包（DMG）内的 clang++ 编译器来编译该应用。对应的命令如下：

```
/Applications/cheerp/bin/clang++ hello_cheerp.cc
-target cheerp
-cheerp-mode=wasm
-cheerp-wasm-loader=hello_cheerp.js
-O3
-o hello_cheerp.wasm
```

不同于 Emscripten，Cheerp 无法直接生成用于运行应用的 HTML 文件，因此我们还需要编写如下 HTML 页面，来将编译生成的 Wasm 模块与 JavaScript “胶水” 脚本文件整合在一起。

```
hello_cheerp.html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Cheerp</title>
  </head>
  <body>
    <script src="hello_cheerp.js"></script>
  </body>
</html>
```

最后，我们在本地启动一个 HTTP 静态服务器，即可在浏览器中运行该应用。

通过这个简单的示例，我们对如何基于 Cheerp 来开发 WebAssembly 应用有了一个基本的认识。限于篇幅，这里不再深入介绍其内部各个组件功能的具体使用方法，感兴趣的读者可以参考官方文档来进一步了解。从总体上看，相较 Emscripten 工具链而言，Cheerp 的出现时间较晚，其所支持的功能较少，并且两者的定位也并不相同。Emscripten 的定位是可以在不修改或少量修改源代码的情况下，将原有的 C/C++ 应用直接转译到 Web 平台上来运行；而 Cheerp 则完全自定义了一套新的规则体系，专门用于使用 C/C++ 语言来编写 Web 应用，并且不向下兼容。

更为重要的一点是，Cheerp 并不是基于 MIT 协议开源的，这就意味着如果要将其用于商业用途，则需要购买 Cheerp 的商用所有权证书，如图 8-14 所示。

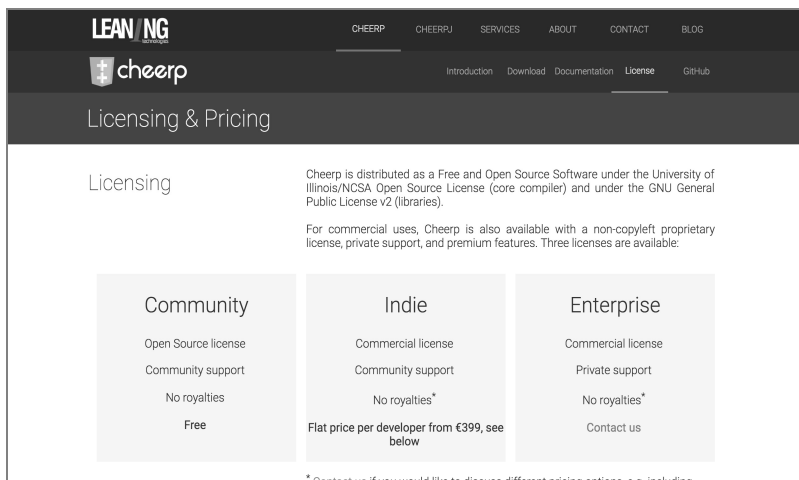


图8-14 Cheerp的商用所有权证书

8.2.2 Webpack 4

对于 Webpack 大家应该都十分熟悉，其在整个前端工程化领域是必不可少的重要组件之一。值得一提的是，在 2018 年 2 月发布的 Webpack 4 版本中，官方首度宣布正式支持对项目中使用到的 Wasm 模块进行默认的 import（导入）/export（导出）过程，如图 8-15 所示。这无疑对 Wasm 技术的普及，以及现代前端开发领域中的工程化整合有着巨大的推动作用。

下面我们将通过一个简单的例子来探索 Webpack 4 中有关 Wasm 的新特性。在通过 Webpack 进行应用打包之前，首先需要使用 Emscripten 来编译生成一个应用在前端项目中的 Standalone 类型的 Wasm 模块。该模块的 C/C++ 代码如下：

```
module.cc
```

```
#include <emscripten.h>
```

```
extern "C" int EMSCRIPTEN_KEEPALIVE add (int a, int b) {
    return a + b;
}
```



图8-15 Webpack 4版本将支持Wasm模块的默认导入/导出

当模块编译完成后，我们还需要准备另外三个代码文件。其中一个是在 Webpack 在编译应用时需要使用的 HTML 模板文件。通过 `HtmlWebpackPlugin` 插件，Webpack 可以在编译过程中直接将生成的 JavaScript 脚本以“script”标签的形式插入 HTML 模板文件中，以输出最终可用于生产环境的首页入口文件。

```
index.html
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Webpack4</title>
  </head>
  <body>
  </body>
</html>
```

第二个是虚拟的上层业务代码文件。这里需要通过 `require.ensure` 以异步的方式来导入 Wasm 模块。并且我们在这里调用了从模块中导出的函数。

```
app.js
```

```
require.ensure([], function(require) {
  let _WASM_MODULE = require('./module.wasm');
  console.log(_WASM_MODULE['_add'](1, 2));
});
```

最后一个就是用于指示 Webpack 进行应用构建的 `webpack.config.js` 配置文件，其代码如下：

```
webpack.config.js
```

```
const HtmlWebpackPlugin = require('html-webpack-plugin');
```



```

module.exports = {
  entry: './app.js',
  mode: 'production',
  output: {
    path: __dirname + '/dist',
    filename: 'app@production.js'
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: __dirname + '/index.html'
    })
  ]
};

```

当一切都准备就绪后，我们便可以在应用的根目录下通过运行 `webpack` 命令来构建该应用了。从现阶段来看，Webpack 对 Wasm 的支持也仅停留在默认的加载与实例化过程上，而对于经由 Emscripten 或 Cheerp 构建的 Wasm 模块来说，由于“胶水”脚本的存在导致暂时还难以将其与 Webpack 本身进行完美的整合。另外，在笔者写下这段文字时，仍然没有在官方文档中找到任何与“怎样初始化 Wasm 模块导入段”相关的话题。因此，当我们尝试通过 Webpack 来打包一个内部含有“import”段结构的模块文件时，可能会发生如图 8-16 所示的错误。



```

→ Webpack git:(master) X webpack
Hash: 7d8ecca092f13a64afc5
Version: webpack 4.15.1
Time: 440ms
Built at: 2018-07-08 10:21:47
3 assets
[0] ./app.js 133 bytes {1} [built]
[1] ./module.wasm 67 bytes {0} [built]

ERROR in ./module.wasm
Module not found: Error: Can't resolve 'env' in '/Users/jason/Desktop/Repo/Book-DISO-WebAssembly/Webpack'
 @ ./module.wasm
 @ ./app.js
Child html-webpack-plugin for "index.html":

```

图8-16 Webpack 4暂时并不支持导入Wasm模块的“import”段结构

提示：笔者所使用的 Webpack 为 4.15.1 版本。

8.2.3 Go 和 Rust 的 WebAssembly 实践

随着 WebAssembly 的发展及其影响力的逐渐扩大，已经有越来越多强类型的主流编程语言开始支持将其源代码编译到 WebAssembly。本节我们将要介绍的便是现阶段十分流行的两种服务端/系统级开发语言，即 Go 和 Rust 在 WebAssembly 技术上的简单实践。这里不会讨论这两

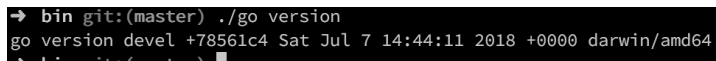
种编程语言在各自语法及应用场景上的特性，而是通过简单的示例来介绍如何将这两种语言的源代码编译成 Wasm 二进制模块，并在浏览器中进行使用。

Go

我们首先编写一段简单的 Go 代码，如下所示。

```
hello_go.go
package main
// 导入 fmt 包，用于格式化输出
import "fmt"
// 定义数学求和函数
func add(a, b int) int {
    return a + b
}
// 主函数
func main() {
    fmt.Println("Hello, WebAssembly!")
    fmt.Println(add(10, 20))
}
```

由于当前以正式版本发布的 Go 语言编译器并没有完全整合编译到 WebAssembly 目标的相关实现，因此，这里需要从 Github 上克隆当前主分支中的最新源代码，并在本地环境中手动编译正处于开发中的“devel”版本编译器。当编译过程结束后，我们可以通过“go version”命令来查看当前 Go 编译器的版本，如图 8-17 所示。



```
➔ bin git:(master) ./go version
go version devel +78561c4 Sat Jul 7 14:44:11 2018 +0000 darwin/amd64
➔ bin git:(master)
```

图8-17 查看当前Go编译器的版本

现在我们通过如下命令将上面的代码编译为 Wasm 模块。这里需要使用“GOARCH”和“GOOS”两个环境变量来设置编译过程的目标架构与系统类型，而此处处在命令中使用的 js/wasm 配置参数则正对应着 WebAssembly 虚拟的目标指令集架构。

```
GOARCH=wasm
GOOS=js
./go build hello_go.go
-o hello_go.wasm
```

当编译完成后，我们会在当前文件夹内发现 Go 编译器生成的 Wasm 二进制模块文件。不同于 Emscripten 所构建的 Standalone 类型的模块，我们在使用该模块时还需要通过配合对应的 JavaScript “胶水”脚本文件，来初始化在进行语言特性绑定时 Go 编译器生成的一些底层系统

调用和其他相关资源。该文件可以在 Golang 的官方 Github 仓库中找到,地址是 https://github.com/golang/go/tree/master/misc/wasm/wasm_exec.js。接下来,我们需要编写一个 HTML 文件,将上面提到的所有资源整合在一起。代码如下:

```
index.html
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Go && WebAssembly</title>
</head>

<body>
  <script src="wasm_exec.js"></script>
  <script>
    const go = new Go();
    WebAssembly.instantiateStreaming(
      fetch("hello_go.wasm"),
      go.importObject
    ).then(result => {
      // 运行main函数
      go.run(result.instance);
    });
  </script>
</body>
</html>
```

可以看到,我们在 HTML 文件中通过 JavaScript 代码实例化 Wasm 模块,以及调用从模块中导出的 main 函数时,都需要借助“胶水”脚本文件中 Go 对象所提供的相关辅助方法来实现。需要注意的是,现阶段通过 Go 编译器编译生成的 Wasm 二进制文件其体积都非常大,如果使用 `wasm-dis` 工具将其转译为对应的 WebAssembly 可读文本代码,那么所生成的整个 WAT 文件将会包含大约 90 万行的代码。可见,这是一个明显需要被优化的地方。

Rust

我们再来看 Rust 语言的表现如何。同样的,我们首先编写一段简单的 Rust 代码,如下所示。

```
hello_rust.rs
fn main() {
  println!("Hello, WebAssembly!");
}
```

```
println!("{}", add(10, 20));  
}  
  
fn add(x: i32, y: i32) -> i32 {  
    x + y  
}
```

然后在确保已经通过“rustup target add wasm32-unknown-emsripten”命令为 Rust 编译器添加了针对 WebAssembly 的编译目标类型的情况下，直接通过如下命令即可编译该应用。

```
rustc hello_rust.rs  
--target=wasm32-unknown-emsripten  
-o hello_rust.html
```

不同于 Go 语言需要在源代码编译完成后再编写对应的 HTML 文件，并绑定 JavaScript “胶水”脚本，Rust 编译器首先将其源代码直接编译为基于平台类型的“.o”对象文件，然后再通过间接调用 Emscripten 工具链的编译器入口脚本 emcc，对这些处于中间状态的对象文件进行转译并生成最终的 Wasm 应用。因此，在执行上面的编译命令时，也请确保 emcc 命令能够在全局环境中被调用。

总的来看，现阶段，将基于 Go 语言编写的应用编译到 Wasm 模块的整体复杂度较高，且目标应用的运行效率较低；而 Rust 则与之相反，基于 Emscripten 工具链的二次编译能够最大程度地优化整个目标应用的资源体积大小以及运行效率。

8.2.4 Binaryen

与 Emscripten 一样，Binaryen 也是由 Mozilla 工程师 Alon Zakai 开发的一套专门用于 WebAssembly 的基础设施工具链。不同于 Emscripten 专注于从 C/C++源代码到上层 Web 应用的构建过程，Binaryen 则更加侧重于处理从 WebAssembly 模块到中间代码（ASM.js）或 IR（Binaryen-IR）的相关事情。作为一套工具链，其在内部提供了如下一些常用的命令行工具。

- wasm-shell: 直接加载并解释执行一段 WebAssembly 可读文本（WAT）代码。
- wasm-as: 将 WebAssembly 可读文本代码转译为对应的二进制模块。
- wasm-dis: 将 WebAssembly 二进制模块解构为对应的可读文本代码。
- wasm-opt: 基于 Binaryen-IR 优化器对一段 WebAssembly 代码（二进制形式/可读文本）进行优化。

- `asm2wasm`: 将 `ASM.js` 代码转译为对应的 WebAssembly 二进制模块。
- `wasm2asm`: 将 WebAssembly 二进制模块转译为对应的 `ASM.js` 代码。
- `wasm-merge`: 用于合并多个 WebAssembly 二进制模块。
- `wasm.js`: 一个独立的 JavaScript 组件库。其中包含了以 JavaScript 代码实现的 Wasm 解释器、`asm2wasm` 命令及 S-表达式解析器等工具。一般可将其作为“polyfill”，为不支持 WebAssembly 特性的浏览器提供使用 JavaScript 模拟的上层 Wasm 虚拟机环境。

8.2.5 WasmFiddle

对于 WasmFiddle 大家应该都比较熟悉了。如图 8-18 所示为 WasmFiddle 平台的初始运行界面。使用 WasmFiddle，可以方便地在线构建简单的 WebAssembly 应用。如果读者是第一次接触 Wasm 技术，想要快速体验并了解该技术的基本实现原理与应用方法，但又不方便在自己的计算机本地环境中安装如 Node.js、Python、Emscripten 等相关的大型构建工具链及其运行时环境，那么使用 WasmFiddle 基于云端构建轻型的 Standalone 类型 Wasm 应用便是一个不错的选择。

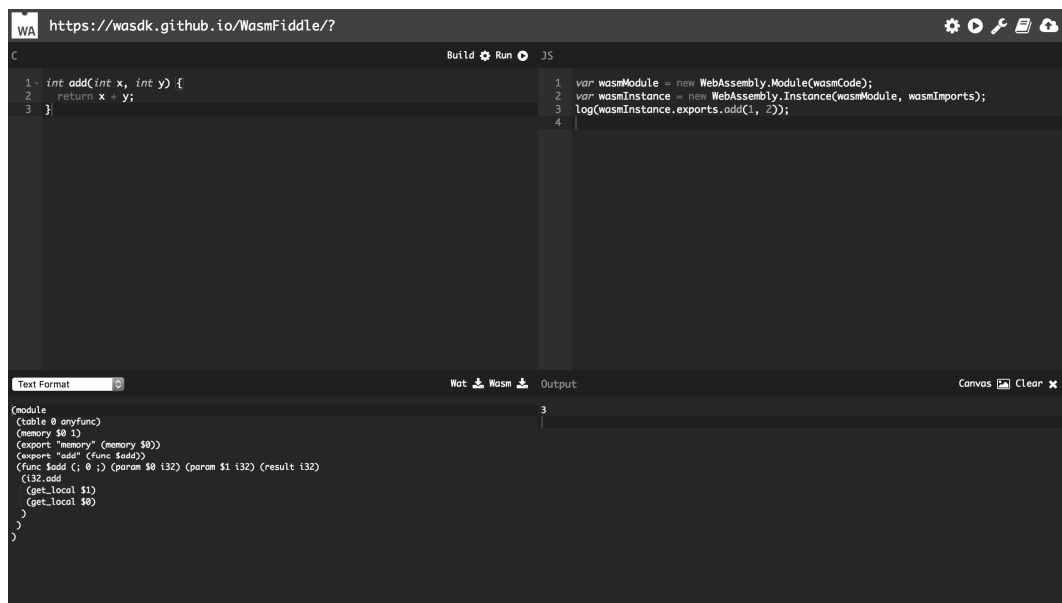


图8-18 WasmFiddle平台的运行界面

8.2.6 Wabt

从所包含的命令行工具的具体功能以及使用方式的角度来看，Wabt 与 Binaryen 工具链十分相似，其在内部也同样提供了很多可以直接使用的命令行工具。但不同的是，相比于 Binaryen 而言，Wabt 则更加关注与底层 WebAssembly 标准格式相关的事情。Wabt 在内部提供了如下命令行工具。

- `wat2wasm`: 将 WebAssembly 可读文本代码转译为对应的二进制模块。
- `wasm2wat`: 将 WebAssembly 二进制模块解构为对应的可读文本代码。
- `wasm-objdump`: 打印与某个 WebAssembly 二进制模块内部结构相关的信息（各段结构的起始地址、段内容，以及对应的虚拟指令集等信息）。
- `wasm-interp`: 解释执行某个 WebAssembly 二进制模块文件，并输出执行路径等相关信息。
- `wat-desugar`: 解析并格式化输出一个 WebAssembly 可读文本格式文件的内容。
- `wasm2c`: 将一个 WebAssembly 二进制模块文件反编译为对应的 C/C++ 源代码。

8.2.7 AssemblyScript

AssemblyScript 是一个专门用于从 TypeScript 到 WebAssembly 的编译器，它在内部使用 Binaryen 将严格类型化的 TypeScript 代码编译成对应的 Wasm 模块。从某种程度来看，这无疑极大地降低了前端开发者构建 WebAssembly 应用的整体难度。关于使用该编译器来构建 Wasm 应用的具体方法可以参考官方文档，由于其使用流程十分简单，这里不再深入介绍。

8.3 WebAssembly 未来草案

在本书的最后一节中，我们将介绍关于 WebAssembly 技术未来发展的相关话题。自从 WWG（WebAssembly Working Group）工作组成立以来，整个 Wasm 官方团队便一直保持着持续的激情，定期召开 WWG 和 WCG 的线上工作视频会议，以保证 Wasm 在后续 Roadmap 中所制定的一些发展计划和新特性能够按时落地。毫无疑问的是，在这些新特性中，每一个特性都能够让 WebAssembly 未来的应用场景及相关生态资源变得越来越丰富。接下来，我们便一起来了解这些正处于实现中的草案。

8.3.1 GC（垃圾回收）

通过为 WebAssembly 提供自动化的 GC（Garbage Collection）实现，使得我们直接在 Wasm 模块中引用并操纵类似于 DOM 和 Web-API 之类的复杂上层 JavaScript 对象成为可能。该草案计划通过引入一种名为“anyref”的引用类型，来不透明地引用位于 Wasm 模块外部的任何复杂对象类型，并隐藏其内部的数据绑定细节。不仅如此，对 GC 特性的支持，还使得 WebAssembly 能够更好地与除 C/C++ 之外的其他静态语言进行关系绑定。

8.3.2 Multi-Thread（多线程）与原子操作

该草案旨在为 WebAssembly 提供类似于 pthread 风格的共享内存管理机制、原子操作符、互斥锁，以及与多线程相关的其他子特性。其中，共享内存管理部分将遵循与 C++ 11 内存模型相兼容的模式进行构建，而在 ECMAScript 8 标准中所提出的基于 SharedArrayBuffer 对象实现的共享内存特性将会被支持。但由于 SharedArrayBuffer 对象存在可能会触发 Spectre 漏洞的风险，因此其逐渐被各主流浏览器禁用，该草案可能会因此而发生改变。

8.3.3 异常处理

在现阶段的 WebAssembly 标准中，当应用发生运行时（Runtime）异常时，对应的错误信息只能在上层浏览器环境中通过 JavaScript 来进行捕获和处理。该草案旨在为 Wasm 标准提供类似 C/C++ 语言中的 try...catch 异常捕获结构，并通过与其相似的底层实现方式来为 Wasm 实现该特性。

8.3.4 多返回值扩展

该草案将允许子程序、特定的指令及块结构在调用完毕或退出其作用域时，可以向当前的数据栈容器中返回多个值。

8.3.5 ES 模块

该草案旨在通过整合 ECMAScript 6 标准中的模块机制，使得我们可以在 HTML 文件中直接通过为“<script type='module'>”标签指定 Wasm 模块资源所在 URL 路径的方式，来加载并实例化对应的模块对象。此时，通过“export”段结构从 Wasm 模块中导出的数据对象，在 ES 6 的模块中将会被作为从该 Wasm 模块中导入（import）的对象。

8.3.6 尾递归

该草案主要用于为 WebAssembly 标准提供尾递归优化特性的具体实现方案。当递归调用是整个函数体中最后执行的语句，并且其返回值不属于表达式的一部分时，我们便称这个递归调用为“尾递归”调用。一般来说，当编译器发现代码中有以尾递归形式组织的递归调用时，会通过复用调用栈帧的操作来优化整个递归调用的执行流程。

8.3.7 BigInts 的双向支持

该草案主要用于为 WebAssembly 标准整合当前在 ECMAScript 标准中最新加入的“BigInt”数据类型，其所代表的是 64 位浮点数特性。在支持该特性后，WebAssembly 会与 ECMAScript 一样将 BigInt 视作一种基本数据类型，并支持在两个环境上下文中进行任意的数据传递与转换。届时，我们也可以使用 BigInt64Array 和 BigUint64Array 这两种类型数组向 Wasm 模块的共享线性内存中写入 BigInt 类型的数据值。

8.3.8 自定义注释语法

在当前 MVP 版本的 WebAssembly 标准中，我们可以通过在 Wasm 二进制模块中设置“Custom”段结构的方式，将任意的自定义元数据存放到模块本身所对应的二进制数据中。比如前面介绍的应用于模块的 SourceMap 地址，便是以这种方式被存放在模块的二进制数据中的。当前存在的问题是，虽然我们可以在模块的二进制数据中使用段结构 ID 值“0”来表示一个实际的“Custom”段类型，但在标准中并没有规定应该以怎样的形式，在模块对应的可读文本（WAT）代码中表示这些自定义的段结构。而该方案正用于解决此问题。

谈到未来，不禁让人充满遐想。互联网技术的发展如此迅速，从最初的使用单一语言做特定领域的事情到现在多语言之间不断地进行融合，技术岗位的相关性与重合度在变大。这一切无不见证着互联网行业与技术的更迭速度之快。但无论未来结果如何，从当下来看，只希望能够实现两件事情：第一，希望借由此书来推动 WebAssembly 技术在国内的普及和使用，让新技术更加快速地进入国内的互联网技术圈，并最终融入具体业务中，甚至推动业务的进一步成长；第二，作为伴随着 Wasm 技术发展而一路走来的人，希望有朝一日也能够为互联网技术的历史长河增添“浓墨重彩”的一笔。